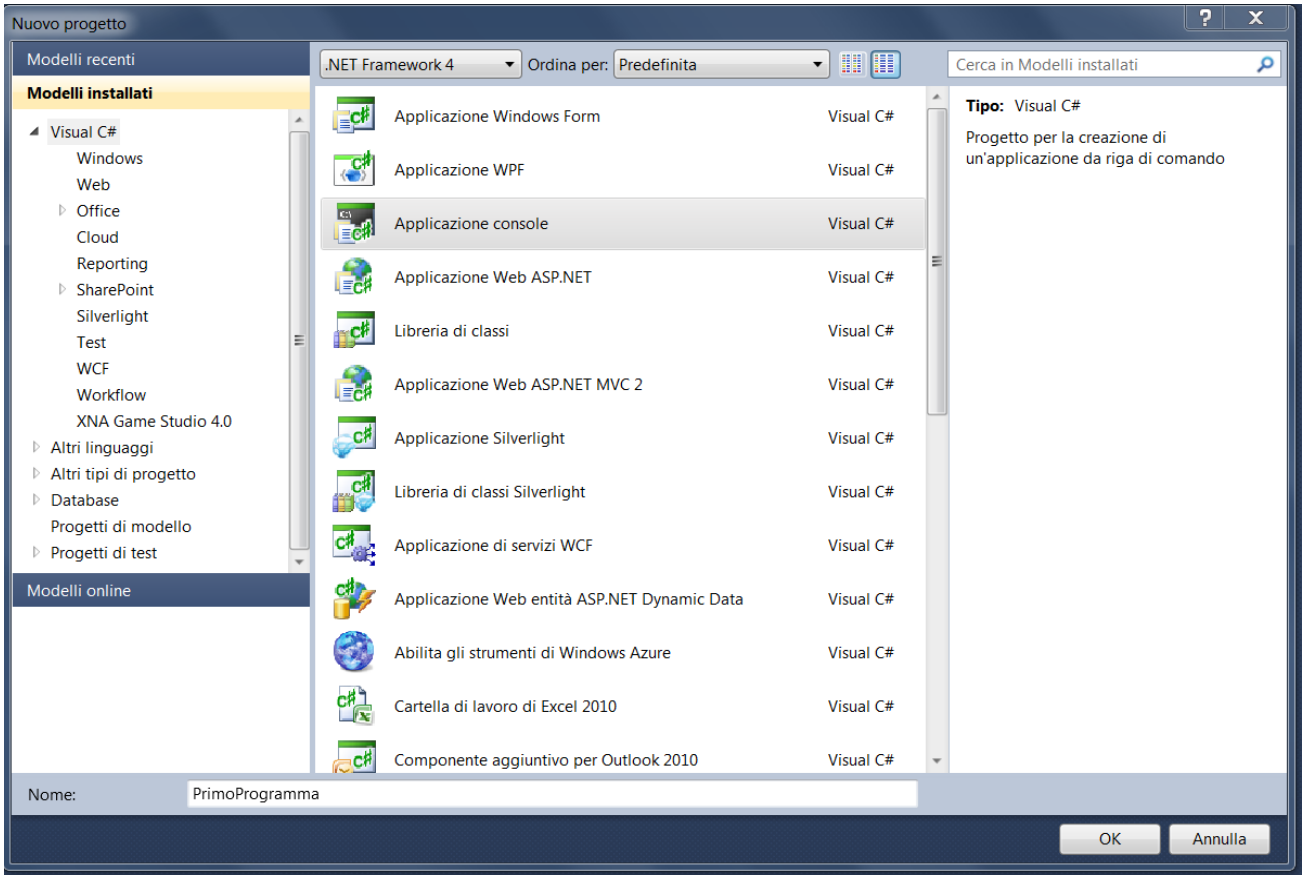
Introduzione al linguaggio di programmazione C#

La sintassi del linguaggio di programmazione **C#** (si legge C sharp) deriva da quella del linguaggio C, ma molti elementi sono presi direttamente dal linguaggio **Java**. Ma se Java è un linguaggio orientato alla semplicità a scapito della velocità di esecuzione, C# unisce alla semplicità e alla rapidità di sviluppo del Java la potenza dei linguaggi C/C++.
C# è concepito per fornire al programmatore C++ più rapidità di sviluppo, senza però ridurne la potenza.
C# resta estremamente fedele a C e a C++ senza allontanarsi per questo da Java.
C# è dunque un linguaggio di programmazione potente, dalla sintassi ricca, adatto allo sviluppo di software orientato agli oggetti per la realizzazione di applicazioni di qualsiasi tipo; non esiste alcuna restrizione del tipo di programmi che è possibile creare.
Per eseguire un programma C# è necessaria la presenza di un particolare ambiente software che si fa carico di gestire l'esecuzione del codice; tale ambiente software è il **.NET Framework**.
.NET Framework è un enorme contenitore di oggetti che svolgono determinati compiti (es. stampa di un documento, calcolo di una funzione matematica, riproduzione di suoni e/o video,etc.).
.NET Framework mette a disposizione un insieme di tecnologie che permettono la compilazione e l'esecuzione di codice C#.

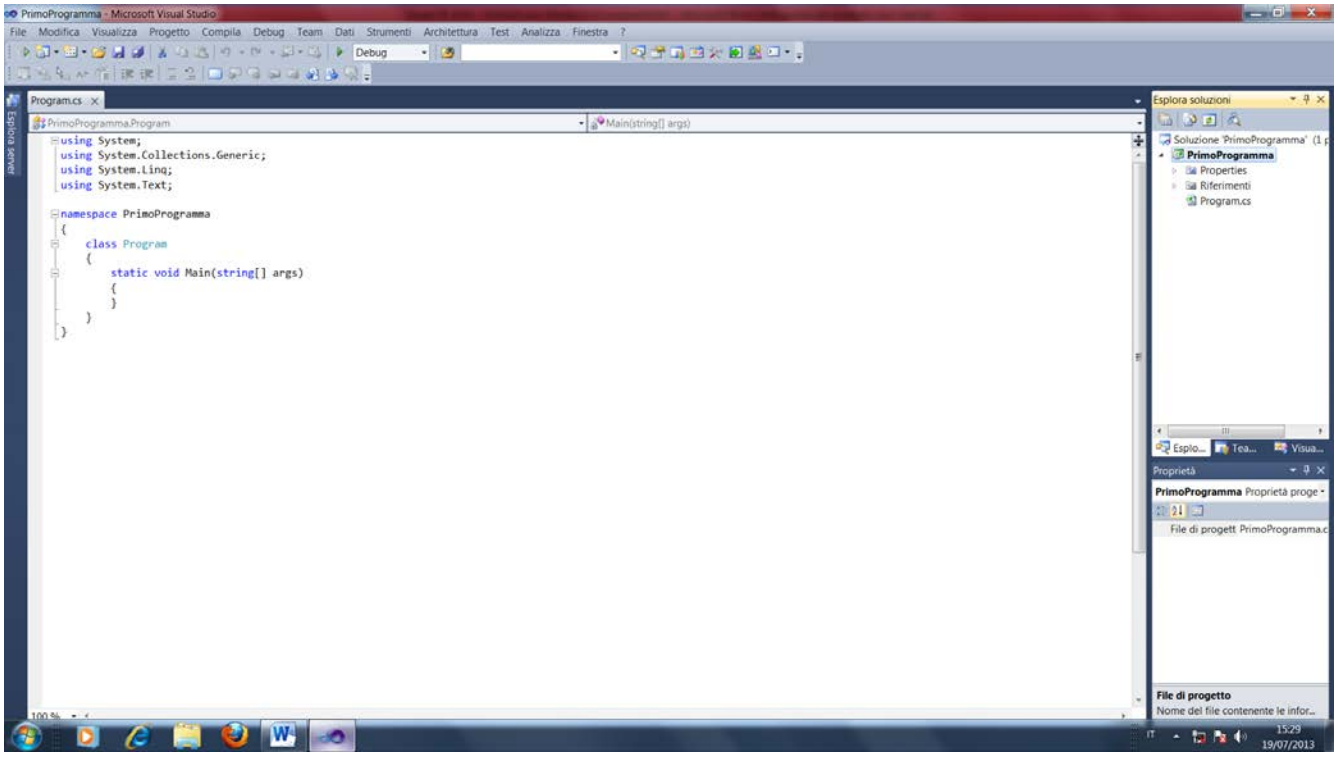
IDE (*Integrated Development Environment*)

Per lo studio del linguaggio C# si farà uso dell'ambiente di sviluppo **IDE** (*Integrated Development Environment*) **Microsoft Visual C# 2010 Express Edition**.
Per creare un programma avviare l'IDE e:

- nella pagina iniziale oppure nel menu **File** selezionare **Nuovo Progetto**
- nella finestra di dialogo **Nuovo Progetto** selezionare **Applicazione console** e assegnare un nome al progetto (ad es. PrimoProgramma), quindi cliccare sul pulsante **OK**.

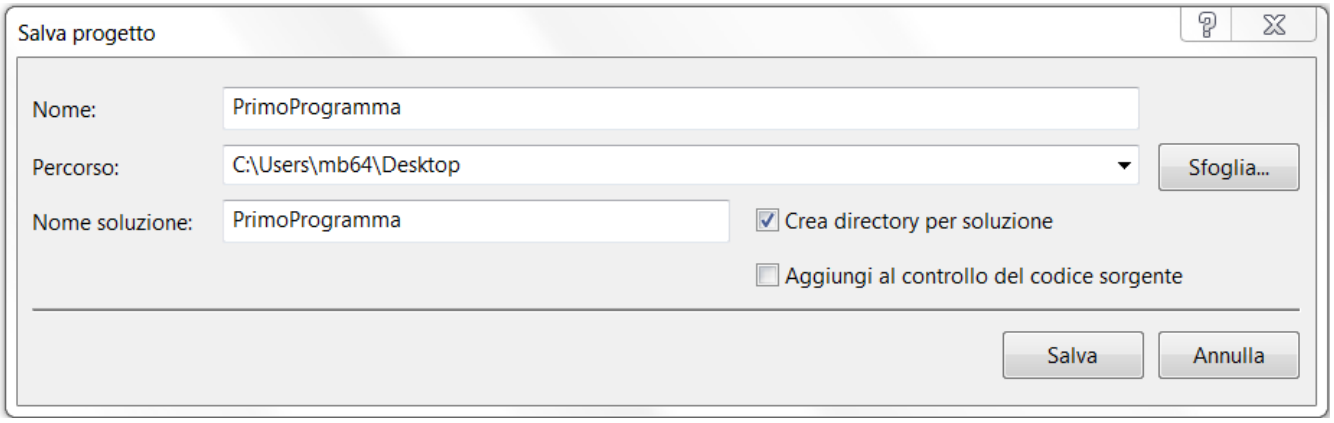


A questo punto compare la finestra sottostante che riporta, nella barra del titolo, il nome del progetto appena creato.



Per salvare il progetto PrimoProgramma appena creato:

- dal menu **File** selezionare **Salva tutto**



Nella finestra di dialogo **Salva progetto** è possibile selezionare il percorso in cui salvare la cartella **PrimoProgramma** in cui Visual Studio genererà i file e le cartelle che permettono la creazione del file eseguibile PrimoProgramma.exe presente nella cartella **Release** all'interno della directory **PrimoProgramma**.
Il seguente codice, scritto automaticamente dal compilatore, viene memorizzato nel file Program.cs e visualizzato all'interno della scheda omonima :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace PrimoProgramma
{
    class Program
    {
        static void Main(string[] args)
        {

        }
    }
}
```

Aggiungere le seguenti righe di codice all'interno della coppia di parentesi graffe più interne:

```
            // primo programma C#
               Console.WriteLine("Ciao, benvenuti in laboratorio");
```

Il codice risultante diventerà quindi:

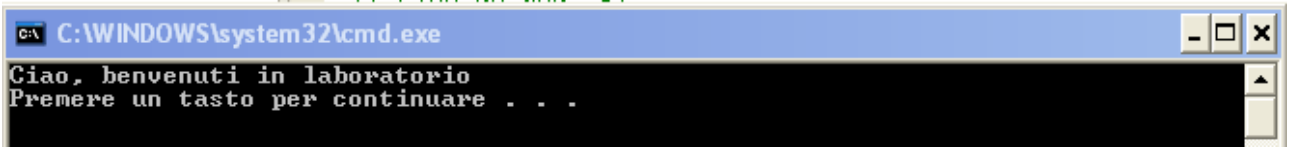
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace PrimoProgramma
{
   class Program
   {
      static void Main(string[] args)
      {
         // primo programma C#
         Console.WriteLine("Ciao, benvenuti in laboratorio");
      }
   }
}
```

.NET Framework è organizzato in **namespace** (spazio dei nomi) e **classi**.
I namespace sono paragonabili a cartelle che costituiscono una raccolta di classi (o di altri namespace) e queste ultime contengono la dichiarazione degli oggetti che possono essere usati in un programma.
Ad es., nel programma precedente si ha il namespace **PrimoProgramma**, la classe **Console** e il **metodo** (funzione) **WriteLine**().
La macchina virtuale del .NET Framework (analogamente a quanto avviene per la *Virtual Machine* di Java) prende il nome di **CLR** (*Common Language Runtime*) e traduce il **codice sorgente** scritto in **C#** in un linguaggio intermedio chiamato *Intermediate Language(IL)* che poi viene convertito in codice macchina dal CLR stesso al momento dell'esecuzione.

Per compilare ed eseguire il programma creato:

- nel menu **Debug** selezionare **Avvia senza eseguire debug**

Compilato il codice sorgente contenuto nel file PrimoProgramma.cs, cioè tradotto il codice sorgente in un codice binario, detto **codice oggetto**, eseguibile dal calcolatore, l'esecuzione del programma mostrerà quanto riportato nella figura seguente:



Per chiudere il progetto :

- dal menu **File** selezionare **Chiudi soluzione**

Per riaprire il progetto PrimoProgramma:

- dal menu **File** selezionare **Apri | Progetto/Soluzione…**
- nella finestra di dialogo **Apri progetto** spostarsi nella cartella del progetto (ad esempio la cartella PrimoProgramma) e selezionare il file con estensione **.sln** (ad esempio il file **PrimoProgramma.sln**). L'estensione **.sln** indica un file di soluzione, dove una soluzione può contenere uno (come nel nostro caso) o più progetti.

Esaminiamo il codice del programma.
La direttiva **using** permette di inserire nel codice il namespace System, il quale consente di usare la classe Console richiamandone un metodo (WriteLine()) che permette di fare la stampa su monitor del messaggio desiderato.
L'utilizzo della direttiva **using** consente di scrivere:

```
Console.WriteLine("Ciao, benvenuti in laboratorio");
            anziché
System.Console.WriteLine("Ciao, benvenuti in laboratorio");
```

Nel codice compaiono anche dei commenti, che il compilatore ignora in fase di generazione del codice eseguibile:

- ***Commenti a riga singola***, che iniziano con **//** e terminano alla fine della riga
- ***Commenti delimitati***, che iniziano con **/*** e terminano con ***/** e possono estendersi su più righe

Il codice del programma inizia con `class Program` che definisce una nuova classe di nome `Program` la quale costituisce il contenitore del programma; si può osservare che l'intero codice è delimitato da una coppia di **{}**.
In realtà il codice realmente eseguito dal computer inizia con:

```
   static void Main(string[] args)
{
        ……….
}
```

Tutti i programmi C# iniziano l'esecuzione con il costrutto precedente: Main() è un metodo, cioè una funzione che a sua volta rappresenta una porzione di codice richiamata tramite il suo nome.
Main() è l'entry point del programma, ossia il punto in cui inizia l'esecuzione del programma ed al suo interno, racchiuse tra **{}** si trovano le uniche vere istruzioni di tutto il programma, istruzioni che devono sempre terminare con il punto e virgola che segnala la fine dell'istruzione stessa.
Nel caso del programma Ciao.cs l'unica istruzione è:

```
Console.WriteLine("Ciao, benvenuti in laboratorio");
```

L'istruzione è formata da una classe che si chiama Console seguita dalla chiamata del metodo (funzione) WriteLine().
Tale metodo viene chiamato passandogli come parametro la stringa di caratteri che si vuole visualizzare sullo schermo ed ha inoltre la funzionalità di andare a capo dopo la visualizzazione.
La classe **Console** fa parte di un insieme di diverse classi contenute nel namespace **System**.
Con la direttiva **using** si è in pratica comunicato al compilatore l'intenzione di utilizzare le classi contenute nel namespace **System**, in particolare la classe **Console** che fornisce il supporto di base per applicazioni in grado di leggere/scrivere caratteri da/su la console.

Le parentesi graffe rappresentano un contenitore di istruzioni il cui insieme viene chiamato **blocco di codice.**
Tutti i progetti avranno quindi una struttura del tipo precedente dove, di volta in volta, cambieranno le istruzioni inserite all'interno del metodo Main().
E' possibile rinominare la classe **Program**:

a) Click destro su **Program**
b) Selezionare **Refactor|Rename…**

Definizione e assegnazione delle variabili

Prima di usare una variabile è necessario definirla (dichiarazione) all'interno del codice nel seguente modo:

nometipo nomevariabile;

dove nomevariabile:

- può contenere caratteri, cifre e trattino basso (underscore)
- non può iniziare con una cifra
- non si possono usare parole riservate di C#

Il C# è un linguaggio *case sensitive*, ossia distingue le maiuscole dalle minuscole.
Il C# è un linguaggio di programmazione in cui è obbligatorio dichiarare le variabili.
E' convenzione comune scrivere i nomi delle variabili in minuscolo e quelli delle costanti in maiuscolo.

Esempio

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Variabili1
{
    class Program
    {
        static void Main(string[] args)
        {
            // dichiarazione della variabile di tipo intero
            int numero;
            // assegnazione del valore alla variabile
            numero = 122;
            // stampa del contenuto della variabile
            Console.WriteLine(numero);
        }
    }
}
```

In questo caso il parametro passato al metodo **`WriteLine()`**è una variabile.

Tipi di dato

La dichiarazione di una variabile serve al compilatore per allocare (fornire) l'esatta quantità di memoria RAM richiesta da una variabile di un determinato tipo.
Si parla di allocazione anche per un file o per un qualsiasi oggetto che debba essere memorizzato nella RAM.
I tipi di dato in C# rientrano in quattro categorie:

- **interi**
- **in virgola mobile**
- **decimali**
- **booleani**

Tipo di dato intero

Il C# fornisce otto tipi di dato intero:

- **int e uint**
- **short e ushort**
- **long e ulong**
- **byte e sbyte**

Tipi int e uint

Il tipo **int** occupa 4 byte (32 bit) di memoria e può contenere un numero intero dotato di segno nel range:

-2147483648 ÷ 2147483647

Il tipo **uint** occupa 4 byte (32 bit) di memoria e può contenere un numero intero privo di segno nel range:

0 ÷ 4294967296

- I tipi **short e ushort** sono interi a 16 bit rispettivamente dotati o privi di segno
- I tipi **long e ulong** sono interi a 64 bit rispettivamente dotati o privi di segno

Esempio

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            // dichiaro e inizializzo la prima variabile intera
            int numero1 = 235;
            // dichiaro e inizializzo la seconda variabile intera
            int numero2 = 265;
            //dichiaro la variabile intera che conterrà il risultato
            int somma;
            /* assegno alla variable somma il risultato
            dell'operazione */
            somma = numero1 + numero2;
            // stampa i valori della variabile numero1
            Console.Write("Il primo valore è ");
            Console.WriteLine(numero1);
            // stampa i valori della variabile numero2
            Console.Write("Il secondo valore è ");
            Console.WriteLine(numero2);
            // stampa il messaggio
            Console.Write("La somma = ");
            // stampa il risultato
            Console.WriteLine(somma);
        }
```

```
    }
}
```

Da notare che il metodo **Write()** stampa la stringa ricevuta come argomento senza andare a capo, mentre il metodo **WriteLine()** stampa il contenuto della variabile ricevuta come argomento e poi va a capo.

Tipo byte

Il tipo **byte** occupa un byte (8 bit) di memoria e può contenere un numero intero senza segno nel range:

0 ÷ 255

Numeri in virgola mobile

Il C# fornisce tre tipi di dato in virgola mobile:

- **float**
- **double**
- **decimal**

Tipo di dato double

Occupa 8 byte di memoria e memorizza numeri con la virgola con una precisione da 14 a 16 cifre significative, nel range seguente:

$$\mathbf{-1{,}7.10^{308} \div -5{,}0.10^{-324}} \text{ e } \mathbf{5{,}0.10^{-324} \div 1{,}7.10^{308}}$$

Esempio

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            // dichiarazione della variabile di tipo double
            double numero;
            // assegnazione del valore
            numero = 3.1459265874125375665;
            // stampa della variabile
            Console.WriteLine(numero);
        }
    }
}
```

Tipo di dato bool (booleano)

Occupa 1 byte di memoria e permette di memorizzare i valori booleani **true** e **false**.

Esempio

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            bool flag; // dichiaro la variabile di tipo bool
            flag = true; // assegnazione del valore
            Console.WriteLine(flag); // stampa della variabile
        }
    }
}
```

Costanti

La sintassi per dichiarare una costante è:

const tipo nomecostante = valore;

Se si cerca di modificare il valore di una costante, viene generato un errore in fase di compilazione.

Esempio

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            // dichiarazione e assegnazione di una costante di tipo double
            const double PIgreco = 3.1459;
            // stampa della costante
            Console.WriteLine(PIgreco);
        }
    }
}
```

Visualizzazione delle informazioni in ambiente console

I metodi **Console.Write()** e **Console.WriteLine()** accettano anche alcuni parametri che servono alla formattazione dei risultati.
Uno di questi parametri è il **segnaposto**, che consiste in un indice numerico, a partire da 0, racchiuso tra parentesi graffe.

Esempi

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication3
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            // dichiarazione della variabile di tipo intero
            int numero;
            // assegnazione del valore
            numero = 122;
            // stampa della variabile
            Console.WriteLine("\nLa variabile contiene il valore {0} ", numero);
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            // dichiarazione della prima variabile intera
            int numero1;
            // dichiarazione della seconda variabile intera
            int numero2;
            // dichiarazione della variabile intera che conterrà il risultato
            int somma;
            // assegnazione alle prime due variabili dei rispettivi valori
            numero1 = 235;
            numero2 = 265;
            // dichiarazione alla variable somma del risultato dell'operazione
            somma = numero1 + numero2;
            // stampa dei valori delle variabili
            Console.Write("\nnumero1 è= {0}\nnumero2 è= {1}\nsomma è= {2}\n",
                          numero1, numero2, somma);
        }
    }
}
```

La variabile numero1 ha come segnaposto {0}, la variabile numero2 ha come segnaposto {1} e la variabile somma ha come segnaposto {2}.

Tipo di dato string

E’ un tipo di dato capace di contenere una sequenza di caratteri, una **stringa**, delimitata da una coppia di virgolette.

Esempio

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication3
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            // dichiaro la variabile di tipo string
            string messaggio;
            // le assegno un valore (testo)
            messaggio = "\nCiao, benvenuti in laboratorio";
            // stampa della variabile
            Console.WriteLine(messaggio);
        }
    }
}
```

All'interno di una stringa è anche possibile inserire ***caratteri di escape***, i quali sono caratteri particolari usati nelle istruzioni di stampa per la formattazione dell'output

.

Caratteri	Descrizione
\a	Segnale sonoro (allarme) \u0007
\b	Backspace \u0008
\t	Tabulazione orizzontale \u0009
\r	Ritorno a capo \u000D.
\v	Tabulazione verticale \u000B.
\f	Avanzamento modulo (pagina) \u000C.
\n	Nuova riga (a capo) \u000A.
\'	Virgoletta singola
\"	Virgolette doppie
****	Back slash

Esempi

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            // dichiaro la variabile di tipo string
            string messaggio;
            // le assegno un valore (testo)
            messaggio = "c:\\esercitazioni\\terza Eln";
            // stampa della variabile
            Console.WriteLine(messaggio);
        }
```

```
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            // dichiaro la variabile di tipo string
            string messaggio;
            // le assegno un valore (testo)
            messaggio = @"c:\esercitazioni\terza Eln";
            // stampa della variabile
            Console.WriteLine(messaggio);
        }
    }
}
```

Metodo ReadLine()

Il metodo **ReadLine()** dell'oggetto ***Console*** legge una stringa fino a quando non incontra il carattere CR, che è il codice associato al tasto <INVIO> e che non fa parte della stringa.

Esempio

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            // dichiarazione della variabile di tipo string
            string messaggio;
            // stampa del messaggio
            Console.Write("\nInserisci una frase: ");
// lettura e assegnazione alla variabile di tutto quanto viene digitato da tastiera
            messaggio = Console.ReadLine();
            Console.Write("\n\t");
            // stampa della variabile
            Console.WriteLine(messaggio);
        }
    }
}
```

Concatenazione

Concatenare due o più stringhe significa unirle in un'unica stringa.
Il procedimento di concatenazione più semplice prevede l'uso del carattere **+** .

Esempi

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            // dichiarazione delle variabili di tipo string
            string nome;
            string cognome;
            string nomecognome;
            // stampa del messaggio
            Console.Write("\nInserisci il nome: ");
            // lettura ed assegnazione del nome
            nome = Console.ReadLine();
            // stampa del messaggio
            Console.Write("\nInserisci il cognome: ");
            // lettura ed assegnazione del cognome
            cognome = Console.ReadLine();
// concatenazione delle due stringhe con un carattere 'spazio' usato come separatore
            nomecognome = nome + ' ' + cognome;
            // stampa della variabile
            Console.WriteLine("\nIl mio nome è {0}", nomecognome);
        }
    }
}
```

Conversioni

Anche i numeri con più di una cifra forniti da tastiera sono sequenze di caratteri, quindi stringhe che vanno acquisite da tastiera tramite il metodo **ReadLine**().
C# mette a disposizione metodi che consentono di trasformare una stringa nei corrispondenti valori numerici dei tipi C# validi. Il namespace ***System*** fornisce anche metodi che realizzano tali conversioni. Tali metodi fanno parte dell'oggetto **Convert** e vengono richiamati con la seguente sintassi:

Convert.xxxx(valore da convertire**)**

dove **xxxx** vengono sostituite dall'indicazione del tipo in cui si deve convertire il valore passato come argomento al metodo in questione.
La tabella seguente riporta alcuni dei metodi C# più usati per la conversione.

ToByte	Converte un valore specificato in un valore byte
ToInt32	Converte un valore specificato in un valore int
ToDouble	Converte un valore specificato in un valore double
ToString	Converte un valore specificato nell'equivalente rappresentazione in forma string

Esempi

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            // dichiarazione di tre variabili intere
            int num1, num2, somma;
            // dichiarazione delle variabili di tipo string
            string numero1;
            string numero2;
            // stampa del messaggio
            Console.Write("\nInserisci il primo numero: ");
            // lettura ed assegnazione del primo numero
            numero1 = Console.ReadLine();
            // prima conversione
            num1 = Convert.ToInt32(numero1);
            // stampa del messaggio
            Console.Write("\nInserisci il secondo numero: ");
            // lettura ed assegnazione del secondo numero
            numero2 = Console.ReadLine();
            // seconda conversione
            num2 = Convert.ToInt32(numero2);
            // somma dei due numeri
            somma = num1 + num2;
            // stampa della variabile
            Console.WriteLine("\nLa somma è {0}", somma);
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            // dichiarazione delle variabili double
            double num1, num2, somma;
            // dichiarazione delle variabili di tipo string
            string numero1;
            string numero2;
            // stampa del messaggio
            Console.Write("\nInserisci il primo numero: ");
            // lettura ed assegnazione del primo numero
            numero1 = Console.ReadLine();
            // prima conversione
            num1 = Convert.ToDouble(numero1);
            // stampa del messaggio
            Console.Write("\nInserisci il secondo numero: ");
            // lettura ed assegnazione del secondo numero
            numero2 = Console.ReadLine();
            // seconda conversione
            num2 = Convert.ToDouble(numero2);
            // somma dei due numeri
            somma = num1 + num2;
            // stampa della variabile somma
            Console.WriteLine("\nLa somma è {0}", somma);
        }
    }
```

```
}
```

Operatori aritmetici

Operazione	C#
Addizione	**+**
Sottrazione	**-**
Moltiplicazione	*****
Divisione	**/**
Resto	**%**

Esempio

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication9
{
    class Program
    {
        static void Main(string[] args)
        {
       // dichiarazione e assegnazione alle prime due variabili dei rispettivi valori
            byte numero1 = 25, numero2 = 21, somma;
            /* assegno alla variable somma il risultato
            dell'operazione */
            somma = (byte)(numero1 + numero2);
            // stampa dei valori delle variabili
            Console.Write("\nnumero1 è = {0}\nnumero2 è = {1}\nsomma è = {2}\n",
                          numero1, numero2, somma);

        }
    }
}
```

Si noti che la somma delle due variabili numero1 e numero2, entrambe di tipo **byte**, restituisce, per impostazione predefinita, un dato **int**. Per assegnare la loro somma alla variabile somma di tipo byte occorre effettuare un **casting esplicito**.
E' da osservare, inoltre, che se la somma delle due variabili suddette eccede il massimo valore consentito per il tipo dichiarato (nell'esempio *byte*) il compilatore non segnala alcun errore, ma il risultato appare errato.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            // dichiarazione e inizializzazione delle variabili
            int numero1 = 250, numero2 = 20.25, prod;

// assegnazione alla variable prod del risultato dell'operazione prodotto
            prod = numero1 * numero2;
```

```
            // stampa dei valori delle variabili
              Console.Write("\nnumero1 è= {0}\nnumero2 è = {1}\nil prodotto è = {2}\n",
                            numero1, numero2, prod);
        }
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            // dichiaro la prima variabile
            int numero1;
            // dichiaro la seconda variabile
            int numero2;
            //dichiaro la variabile che conterrà il risultato
            int divis;
            // assegno alle prime due variabili i rispettivi valori
            numero1 = 250;
            numero2 = 20;
            /* assegno alla variable somma il risultato
            dell'operazione divisione */
            divis = numero1 / numero2;
            // stampo i valori delle variabili
            Console.Write("\nnumero1 è= {0}\nnumero2 è= {1}\nil risultato è={2}\n",
                            numero1, numero2, divis);
        }
    }
}
```

Nell'esempio precedente, essendo i **due operandi di tipo intero**, il risultato della divisione è un tipo intero (**int**), mentre se almeno uno dei due operandi è di tipo double, il risultato è di tipo double.
Nell'esempio precedente l'utilizzo di due variabili di tipo intero ha prodotto un **troncamento** della parte decimale; per ottenere il risultato corretto è indispensabile che la variabile contenente il risultato della divisione sia ***double*** e che almeno una delle due variabili che vengono divise sia ***double***.

Esempio

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            // dichiaro la prima variabile di tipo float
            double numero1;
            // dichiaro la seconda variabile
            int numero2;
            //dichiaro la variabile che conterrà il risultato
            double divis; // float
            // assegno alle prime due variabili i rispettivi valori
```

```
            numero1 = 250; // float
            numero2 = 20;
            /* assegno alla variable divis il risultato
            dell'operazione */
            divis = numero1 / numero2;
            // stampo i valori delle variabili
            Console.Write("\nnumero1 è= {0}\nnumero2 è= {1}\nil risultato è={2}\n",
                          numero1, numero2, divis);
        }
    }
}
```

- Per ottenere il resto di una divisione tra interi nel C# è presente l'operatore **%**

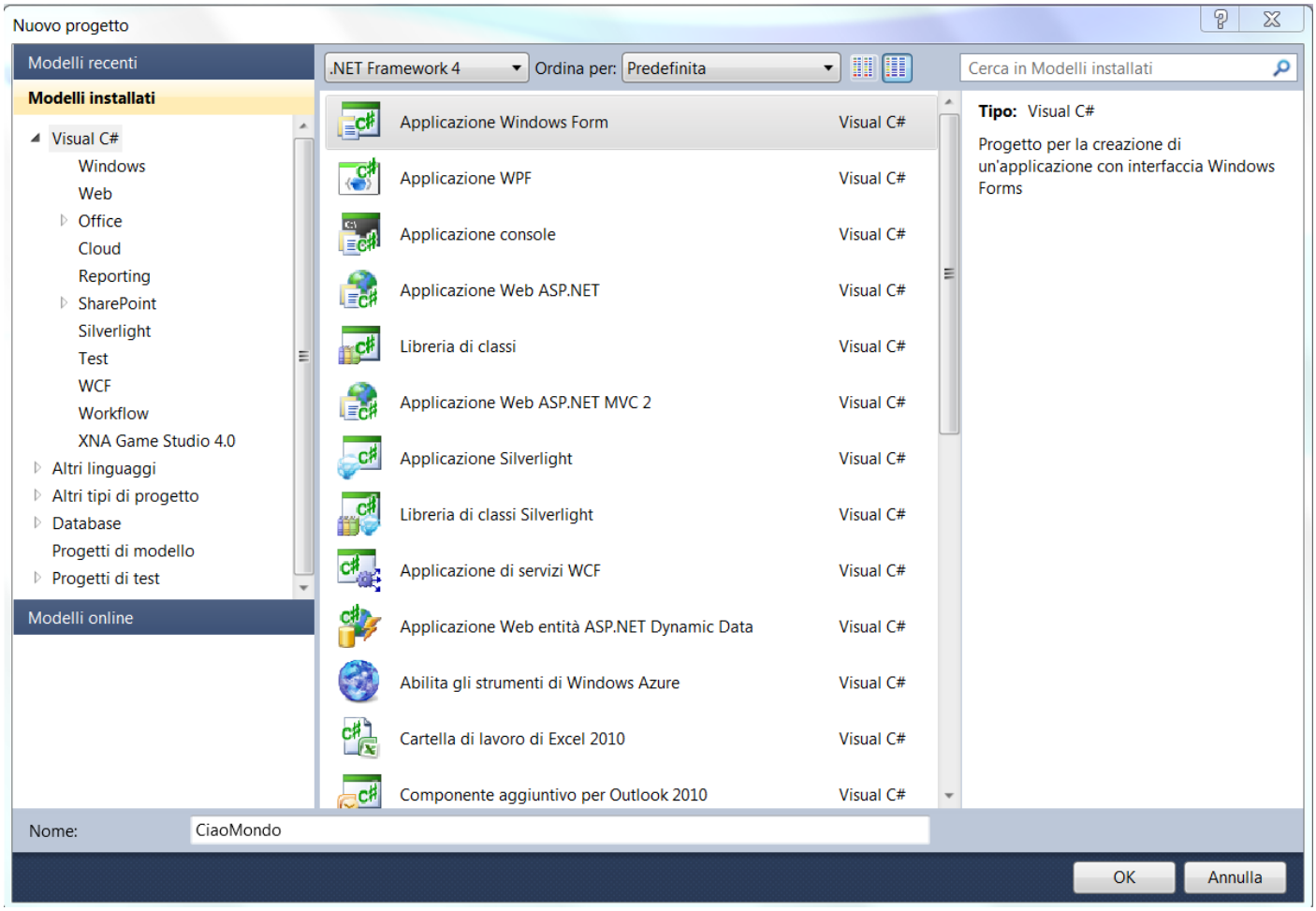
Esempio

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            // dichiarazione delle variabili
            int numero1, numero2, resto;
            numero1 = 25;
            numero2 = 2;
            // assegna alla variable resto il resto della divisione tra interi
            resto = numero1 % numero2;
            // stampa i valori delle variabili
            Console.Write("\nnumero1 è= {0}\nnumero2 è= {1}\nil resto è={2}\n",
                           numero1, numero2, resto);
        }
    }
}
```

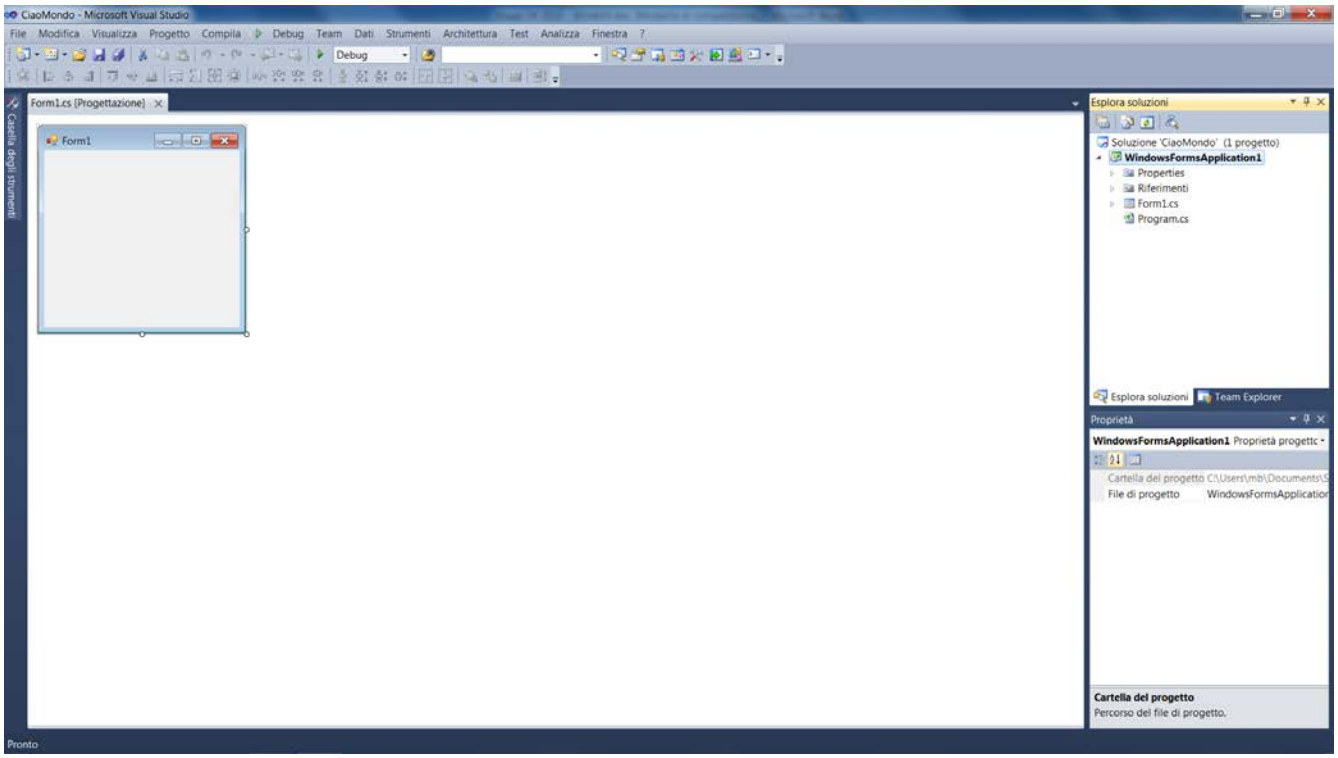
Applicazioni Windows Form

Per creare un'applicazione Windows Form avviare l'IDE e:

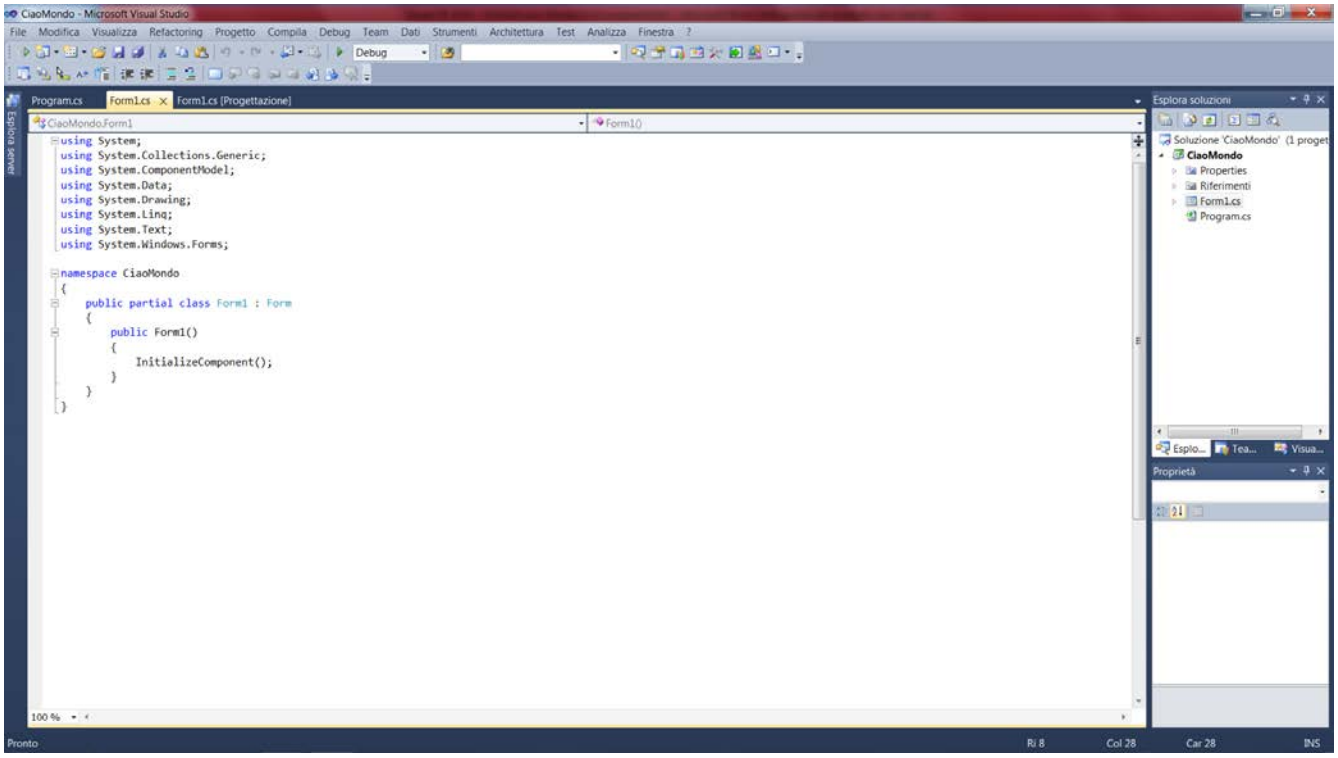
- nella pagina iniziale o nel menu **File** selezionare **Nuovo Progetto**
- nella finestra di dialogo **Nuovo Progetto** selezionare **Applicazione Windows Form** e assegnare un nome al progetto nella parte inferiore della finestra (ad es. CiaoMondo)
- cliccare il pulsante OK



Analogamente a quanto visto per le Applicazioni Console,Visual Studio genera file e cartelle del progetto salvandoli all'interno della cartella CiaoMondo.
La figura seguente mostra una vista dell'applicazione **CiaoMondo** creata dopo aver cliccato sul pulsante **OK** nella finestra di dialogo **Nuovo progetto.**
Ora, senza aver scritto una sola riga di codice, si dispone di un programma Windows funzionante.
Come visto in precedenza, per eseguire il programma dal menu **Debug** scegliere **Avvia senza eseguire debug.** Il risultato è una finestra chiamata **Form1** che non svolge alcuna funzione, ma che possiamo solo chiudere cliccando sulla croce bianca in campo rosso in alto a destra.
Si noti che la finestra **Form1** che appare dopo aver cliccato su **Avvia senza eseguire debug** si trova in stato di esecuzione, mentre la finestra **Form1** visualizzata dopo aver chiuso l'applicazione si trova in stato di progettazione.



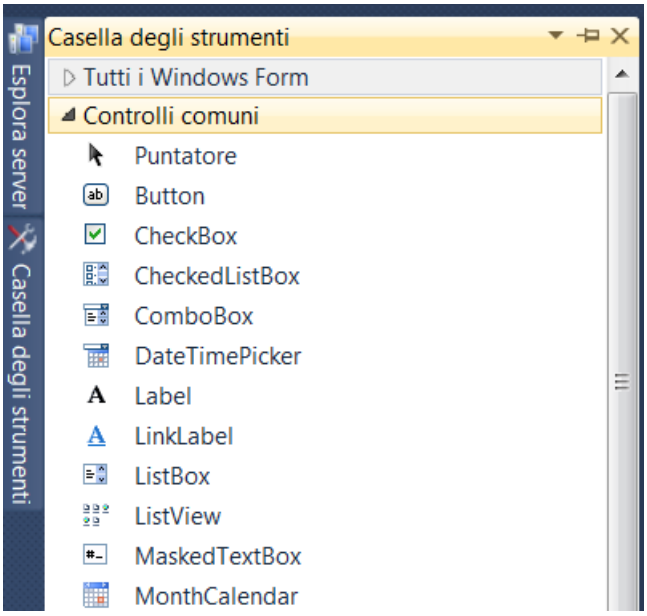
Cliccando col pulsante destro del mouse sulla finestra **Form1** in *modalità Progettazione* e selezionando **Visualizza codice** è possibile passare alla *modalità Vista codice.*



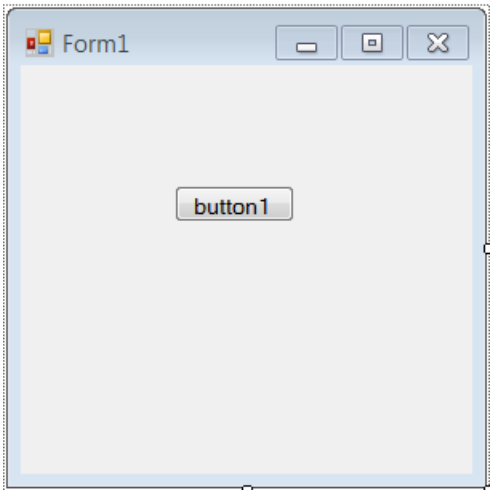
Controlli Button e Label

Torniamo in modalità Progettazione selezionando la scheda **Form1.cs(Progettazione)** e modifichiamo il **Form1** aggiungendo ad esso un pulsante (controllo **button**).

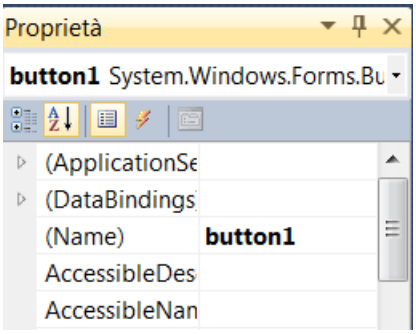
Per visualizzare i controlli che possono essere inseriti in un form Visual C# usa una Casella degli Strumenti che può essere visualizzata selezionando tale voce dal menu **Strumenti** o cliccando sull'icona  nella barra degli strumenti.
La Casella degli strumenti contiene numerosi controlli usati nei form Windows. Per inserire un controllo



Button fare doppio click sul controllo omonimo nella Casella degli strumenti o, in alternativa, singolo click sul controllo trascinandolo sul Form per poi rilasciarlo nel punto desiderato.
E' sempre possibile modificare la posizione del controllo all'interno del Form selezionandolo e trascinandolo nella posizione voluta.
La figura sottostante mostra il Form1 dopo l'inserimento del controllo **Button**.
Per modificare manualmente le dimensioni del controllo **Button** è necessario selezionarlo; verranno visualizzati otto piccoli quadrati in un riquadro che racchiude il controllo Button. E' possibile ridimensionare il controllo spostando il mouse su uno dei quadratini. Il cursore diventa una freccia a due punte: per ridimensionare il pulsante trascinare il cursore tenendo premuto il tasto sinistro del mouse.

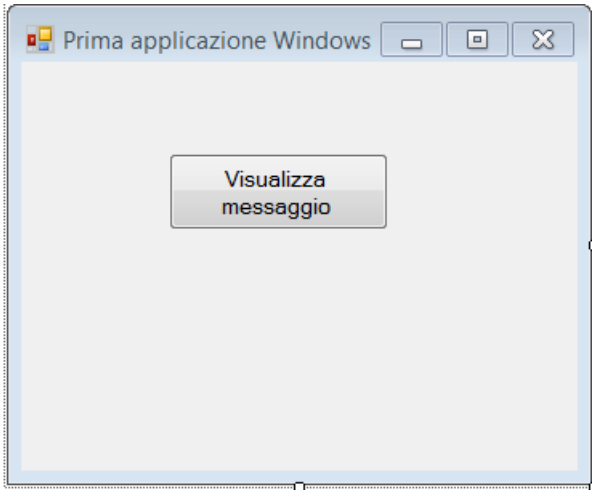


Modifichiamo ora la **proprietà Name** del controllo Button. Tale proprietà è importante perché il suo valore indica il nome con cui si farà riferimento al controllo all'interno del codice.
Per impostazione predefinita il primo pulsante inserito in un Form si chiama Button1, il secondo Button2, il terzo Button3 e così via (analogamente per tutti gli altri controlli).
E' preferibile modificare il nome predefinito assegnandone al controllo uno che ne illustri l'utilizzo. Nel nostro caso, volendo visualizzare sul Form un messaggio mediante un click (in fase di esecuzione) sul pulsante, assegneremo a quest'ultimo il nome (la proprietà **Name**) **btnMessaggio**.
Per modificarela proprietà **Name** del controllo **Button1** in *vista progettazione* scegliere dal menu Visualizza la voce Finestra proprietà oppure cliccare sull'icona  della Barra degli Strumenti.

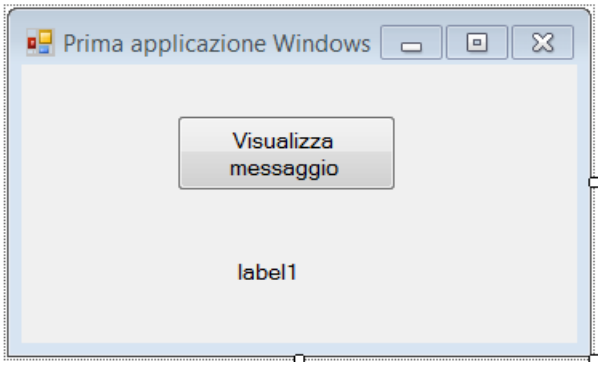


Selezionare(un click solo) il controllo Button1 e poi, nella Finestra proprietà, modificare la proprietà Name cancellando Button1 e sostituendolo con **btnMessaggio**.
Ora vogliamo modificare il testo **button1** che compare sul pulsante sostituendolo con il testo "Visualizza messaggio"; per fare ciò occorre modificare la proprietà **Text** del pulsante.
Analogamente al procedimento adottato in precedenza, selezionare il pulsante, andare nella Finestra proprietà e, nella proprietà **Text**, cancellare button1 e scrivere "Visualizza messaggio".

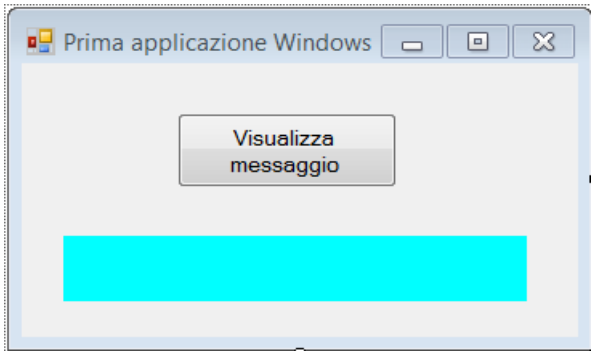
Modificare anche la proprietà **Text** del Form da Form1 a "Prima applicazione Windows".
A questo punto il Form apparirà come nella figura sottostante.



Come si può osservare, la proprietà Text del Form determina il testo visualizzato nella barra del titolo del Form.
A questo punto occorre aggiungere al Form un controllo **Label** (etichetta) che permetta la visualizzazione di un messaggio.
Dalla Casella degli Strumenti, con una procedura analoga a quella vista per il controllo Button, si aggiunge al Form un controllo Label (vedere figura sottostante).
Il controllo Label ha la proprietà Autosize di tipo **bool** che, quando impostata su **True** (impostazione predefinita) ridimensiona automaticamente l'etichetta in modo che possa visualizzare il testo in essa contenuto.
Impostare la proprietà Autosize su **False**, quindi svuotare la proprietà **Text** cancellando **label1** e poi modificarne la proprietà **Name** impostandola su **lblMessaggio**.



Sono anche disponibili, per il controllo Button e per quello Label, le proprietà **BackColor**, **ForeColor** e Font per cambiare, rispettivamente, colore di sfondo, colore del testo visualizzato tipo di carattere utilizzato.
A questo punto il Form si presenta come nella figura in basso.



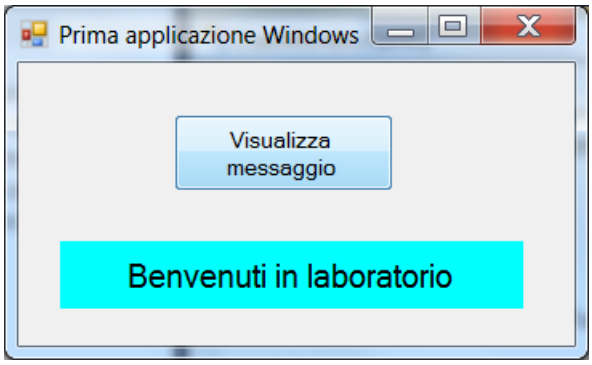
Occorre ora scrivere il codice che sarà eseguito quando, in fase di esecuzione si cliccherà sul pulsante. Le applicazioni Windows, oltre ad avere un aspetto diverso da quelle Console, si comportano anche in modo diverso. Nelle applicazioni Console in cui si usa la programmazione detta *procedurale* le istruzioni vengono eseguite nell'ordine in cui sono state scritte, mentre le applicazioni Windows sono basate sugli ***eventi***, cioè dalle azioni dell'utente.
Nel nostro caso l'evento sarà il click eseguito dall'utente sul pulsante, occorre quindi creare un metodo che venga eseguito quando si verifica tale evento e scrivere al suo interno le istruzioni necessarie a visualizzare un messaggio nella Label.
Per realizzare tale operazione fare doppio click, in *Modalità progettazione*, sul controllo Button di nome **btnMessaggio**, oppure selezionare il pulsante, nella Finestra proprietà cliccare sull'icona  e fare doppio click sull'evento Click.
Verrà creato il seguente metodo che gestirà l'evento click sul pulsante **btnMessaggio**:

```
private void btnMessaggio_Click(object sender, EventArgs e)
{

 }
```

Il codice per visualizzare il messaggio deve essere scritto all'interno della coppia di parentesi graffe:

```
 private void btnMessaggio_Click(object sender, EventArgs e)
 {
    lblMessaggio.Text = "Benvenuti in laboratorio";
 }
```

Avviare il programma e cliccare sul pulsante (vedere figura seguente):

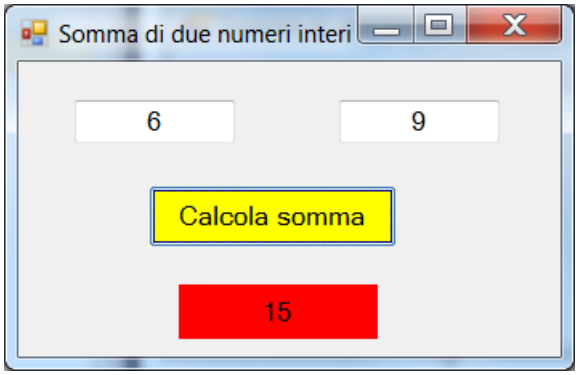


Nel codice è stata assegnata una stringa alla proprietà **Text** (che è di tipo string) del controllo Label di nome **lblMessaggio**.
Per accedere alle proprietà di un controllo all'interno del codice occorre usare ilnome del controllo, digitare il punto e selezionare la proprietà desiderata.
Il nome del metodo, btnMessaggio_Click(), indica che le istruzioni al suo interno saranno eseguite quando si verificherà l'evento click sul pulsante di nome btnMessaggio.

Controllo TextBox

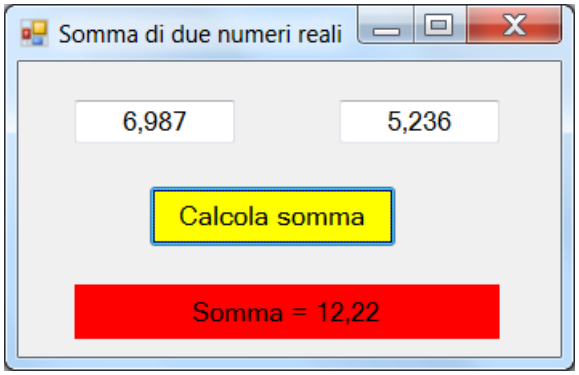
Creare un nuovo progetto e assegnargli il nome SommaNumeri.
Aggiungere al Form1 un controllo Button (rinominarlo come btnSomma), un controllo Label (rinominarlo come lblRisultato) e due controlli **TextBox** (caselle di testo) assegnando loro i nomi txtNum1 e txtNum2.
I controlli TextBox (input) permettono di acquisire i dati che il programma deve elaborare, in questo caso i due numeri interi da sommare, mentre la Label (output) permette solamente di visualizzare il risultato dell'elaborazione, in questo caso la somma dei due numeri.
Assegnare al controllo btnSomma la proprietà Text "Calcola somma" e svuotare la proprietà Text del controllo lblRisultato.
Fare doppio click, in modalità Progettazione, sul controllo btnMessaggio per creare il metodo in cui scrivere le istruzioni per l'acquisizione dei numeri interi dalle caselle di testo e il calcolo della somma:

```
private void btnSomma_Click(object sender, EventArgs e)
 {
    // dichiarazione variabili
    int num1, num2, somma;
    // acqusizione dei numeri dai TextBox
    num1 = Convert.ToInt32(txtNum1.Text);
    num2 = Convert.ToInt32(txtNum2.Text);
    // calcolo della somma
    somma = num1 + num2;
    // visualizzazione del risultato nella Label
    lblRisultato.Text = Convert.ToString(somma);
 }
```



Analizzando il codice si osserva che il contenuto di un TextBox, cioè la sua proprietà Text, prima di essere assegnato ad una variabile int deve essere convertito, essendo la proprietà Text di tipo string.
Analogamente, prima di assegnare il contenuto della variabile somma (di tipo int) alla proprietà Text della Label occorre convertirlo in strig. Eseguire il programma digitando due numeri nei TextBox e cliccare sul pulsante.

Se si vuole realizzare la somma di due numeri reali, il codice diventa il seguente:




```
private void btnSomma_Click(object sender, EventArgs e)
 {
        // dichiarazione variabili
        double num1, num2, somma;
        // acqusizione dei numeri dai TextBox
        num1 = Convert.ToDouble(txtNum1.Text);
        num2 = Convert.ToDouble(txtNum2.Text);
        // calcolo della somma
        somma = num1 + num2;
        // visualizzazione del risultato nella Label
lblRisultato.Text = string.Format("Somma = {0:f2}", somma);
        }
```

Il metodo **Format** della classe string ha una sintassi analoga a quella dei metodi Write/WriteLine della classe **Console** e permette di visualizzare il risultato con due cifre dopo la virgola.

Istruzione di selezione if

Sintassi:

```
if(condizione)
   istruzione_true1;
else if
   istruzione_true2;
      ……….
 else   // facoltativa
    istruzione_false;
```

E' possibile sostituire a istruzione_true e/o a istruzione_false un ***blocco di codice*** composto da due o più istruzioni racchiuse tra **{ }**.

L'istruzione di selezione **if** è di tre tipi, a seconda del numero di blocchi di codice alternativi:

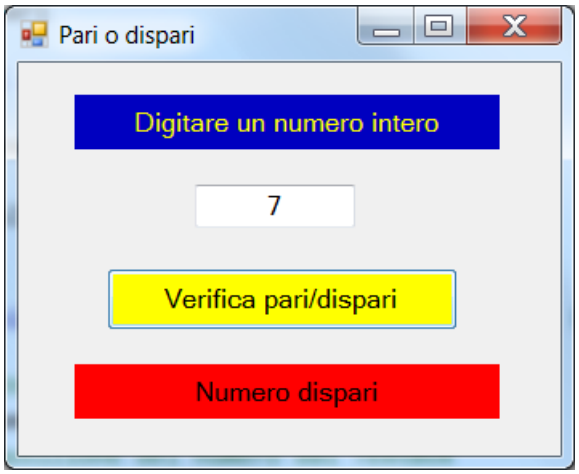
- si utilizza la dichiarazione **if** se si desidera che sia eseguito un blocco di codice se una certa condizione è vera (**true**), mentre si desidera che tale blocco di codice non venga eseguito se la condizione è falsa (**false**)
- si utilizza la dichiarazione **if…else** se si desidera che sia eseguito un blocco di codice se una certa condizione è **true**, mentre si desidera saltare l'esecuzione di tale blocco ed eseguire un secondo blocco se la condizione è **false**. Spesso questa struttura è usata quando sono presenti due alternative, quali si o no, pari o dispari, maschio o femmina, testa o croce etc.
- la dichiarazione **if…else…if** è simile alla dichiarazione **if…else**, ad eccezione del fatto che viene usata quando esistono più di due alternative. Per esempio un numero può essere nullo, positivo o negativo, dal lancio di un dado può uscire uno tra sei possibili numeri etc.

Nell'istruzione di selezione **if** sono spesso utilizzati gli **operatori relazionali**, essi compaiono all'interno della condizione da testare.

Operatore	Descrizione
==	Uguale a
<	Minore di
>	Maggiore di
<=	Minore o uguale a
>=	Maggiore o uguale a
!=	Diverso da

Esempio di istruzione if…else

Applicazione Windows che richiede la digitazione di un numero intero e visualizza un messaggio per indicare se è pari o dispari.
Creare un progetto con nome PariDispari, quindi aggiungere al Form un TextBox, un Button e due Label.



Per quanto riguarda la Label con colore di sfondo blu, non è necessario cambiarne la proprietà Name (nel codice non accederemo a tale etichetta), è sufficiente modificarne la proprietà Text per visualizzare l'indicazione "Digitare un numero intero" da fornire all'utente.
Alla Label con sfondo rosso assegnare il nome **`lblRisultato`** e svuotare la proprietà Text.
Al controllo Button assegnare il nome **`btnPariDispari`** e modificare la proprietà Text in modo da visualizzare sul pulsante il testo mostrato nella figura a fianco.
Al controllo TextBox assegnare il nome **`txtNumero.`**
In progettazione doppio click sul pulsante e, nel metodo creato, scrivere il seguente codice:

```
private void btnPariDispari_Click(object sender, EventArgs e)
 {
    // dichiarazione variabile
    int numero, resto;
    // acqusizione del numero dal TextBox
    numero = Convert.ToInt32(txtNumero.Text);
    // divide numero per 2 ricavando il resto
    resto = numero % 2;
    // verifica valore resto e visualizza pari o dispari
    if(resto == 0)
       lblRisultato.Text = "Numero pari";
    else
       lblRisultato.Text = "Numero dispari";
 }
```

Come già visto, l'operatore % restituisce il resto di una divisione e tale resto viene salvato nella variabile omonima.
All'interno dell'istruzione **if** l'operatore relazionale == confronta il contenuto della variabile resto con la costante numerica zero: se la condizione è true (vera) viene eseguita l'istruzione sottostante l'if e saltata quella sottostante l'**else**, se invece la condizione è false (falsa), cioè la variabile resto contiene un numero diverso da zero, allora viene saltata l'istruzione sottostante l'if ed eseguita quella sottostante l'**else**.
L'applicazione ha un problema: se si clicca il pulsante prima di aver inserito un numero nel TextBox compare un messaggio di errore e bisogna chiudere il programma.
Per evitare questo inconveniente utilizziamo la prima forma di istruzione **if** modificando il codice nel modo seguente:

```
private void btnPariDispari_Click(object sender, EventArgs e)
 {
    // dichiarazione variabile
    int numero, resto;
    // verifica se il TextBox contiene una stringa vuota
    if (txtNumero.Text == "")
       return;
    // acqusizione del numero dai TextBox
```

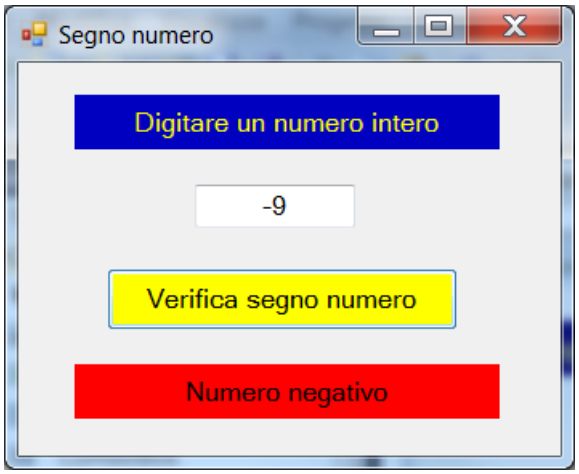
```
        numero = Convert.ToInt32(txtNumero.Text);
        // divide numero per 2 ricavando il resto
        resto = numero % 2;
        // verifica valore resto e visualizza pari o dispari
        if(resto == 0)
            lblRisultato.Text = "Numero pari";
        else
            lblRisultato.Text = "Numero dispari";
    }
```

Con l'istruzione **if** (txtNumero.Text == "") verifichiamo se il controllo TextBox contiene una stringa vuota: se la condizione è vera, viene eseguita l'istruzione **return** che termina l'esecuzione delle istruzioni all'interno del metodo btnPariDispari_Click(), saltando quindi l'esecuzione delle istruzioni sottostanti return. Se invece la condizione all'interno dell' **if** è falsa viene saltata l'istruzione **return** ed il programma prosegue normalmente.

Esempio di istruzione if…else…if

Applicazione Windows che richiede la digitazione di un numero intero e visualizza un messaggio per indicare se è pari a zero, positivo o negativo.

Creare un progetto con nome SegnoNumero, quindi aggiungere al Form un TextBox, un Button e due Label.



Assegnare le proprietà Name seguenti:
TextBox: txtNumero
Button: btnTest
Label: lblNumero

In progettazione doppio click sul pulsante e, nel metodo creato, scrivere il seguente codice:

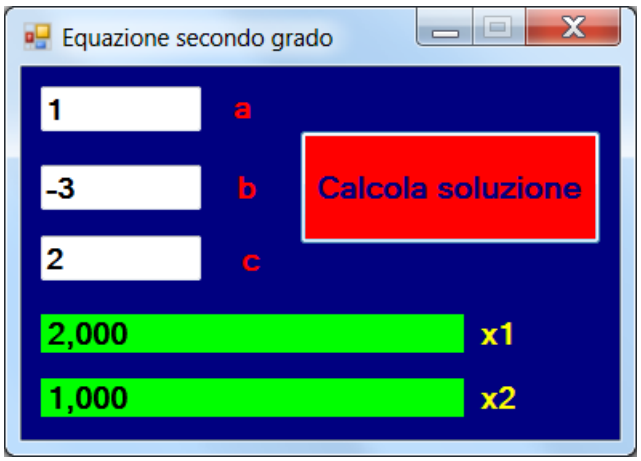
```
private void btnTest_Click(object sender, EventArgs e)
    {
        int numero; // dichiarazione variabile
        // acquisizione del numero dai TextBox
        numero = Convert.ToInt32(txtNumero.Text);
        if (numero == 0)            // verifica se numero è pari a zero
            lblNumero.Text = "Numero nullo";
        else if(numero > 0)
            lblNumero.Text = "Numero positivo";// verifica se numero è positivo
        else
            lblNumero.Text = "Numero negativo";
    }
```

All'interno dell'istruzione **if** l'operatore relazionale == confronta il contenuto della variabile numero con la costante numerica zero: se la condizione è true (vera) viene eseguita l'istruzione sottostante l'if e vengono saltate le istruzioni sottostanti l'**else… if** e l'**else**, se invece la condizione è false (falsa), cioè la variabile numero contiene un valore diverso da zero, allora viene testata la condizione all'interno dell'istruzione **else… if**. La condizione all'interno dell'istruzione **else… if** verifica se la variabile numero contiene un valore maggiore di zero: se la condizione è vera viene eseguita l'istruzione lblNumero.Text = "Numero positivo" e saltata l'istruzione sottostante l'**else**, se invece è falsa viene saltata l'istruzione sotto l'**else… if** ed eseguita l'istruzione lblNumero.Text = "Numero negativo".

Operatori logici

Operatore	Descrizione
&&	AND (e)
||	OR (oppure)
!	NOT (negazione)
^	XOR (or esclusivo)

A volte, affinché possa aver luogo un'azione è necessario che due confronti generino un risultato true. Per esempio, una persona può votare solo se è maggiorenne e (**AND**) se ha la cittadinanza del paese in cui si svolgono le votazioni; entrambe le condizioni devono essere vere affinché la persona sia ammessa al voto
Al contrario, in altre situazioni affinché possa aver luogo un'azione è sufficiente che uno solo tra due confronti generi un risultato true. Per esempio, per essere ammesso all'esame di Stato, il candidato deve aver frequentato la classe quinta con profitto oppure (**OR**) aver ottenuto al termine della classe quarta una media dei voti superiore all'otto.
Come esempio di utilizzo dell'operatore logico AND (**&&**) vediamo un programma che ricavi le radici dell'equazione di secondo grado $ax^2 + bx + c = 0$.
Creare un progetto e salvarlo con nome **EqSecondoGrado**.



Al Form vengono aggiunti un controllo Button, tre TextBox (per la digitazione dei coefficienti a, b, c) e sette Label, di cui due (quelle con sfondo verde) dedicate alla visualizzazione delle due soluzioni x1 e x2.
Assegnare le proprietà Name seguenti:
TextBox: txta, txtb, txtc
Label: lblX1, lblX2
Button: btnCalcola

In progettazione doppio click sul pulsante e, nel metodo creato, scrivere il seguente codice:

```
private void btnCalcola_Click(object sender, EventArgs e)
 {
    // dichiarazione variabili
    double a, b, c, delta, x1, x2;
    // acquisisce coefficienti dai TextBox
    a = Convert.ToDouble(txta.Text);
    b = Convert.ToDouble(txtb.Text);
    c = Convert.ToDouble(txtc.Text);
    // verifica se l'eq. è impossibile
    if (a == 0 && b == 0)
    {
       lblX1.Text = "Equazione impossibile";
       lblX2.Text = ""; // svuota lblX2 assegnandole una stringa vuota
    }
    // verifica se l'eq. è di primo grado
    else if (a == 0)
    {
       // visualizza soluzione con tre cifre dopo la virgola
       lblX1.Text = string.Format("{0:f3}", -c / b);
       lblX2.Text = "";
    }
```

```
        else //soluzione eq.secondo grado
        {
            // calcola delta = b*b-4ac con metodo Pow(base, esponente) della
            // classe Math per il calcolo dell'esponenziale
            delta = Math.Pow(b, 2) - 4 * a * c;
            // ricava radici equazione con metodo Sqrt() della
            // classe Math per il calcolo della radice quadrata
            x1 = (-b + Math.Sqrt(delta)) / (2 * a);
            x2 = (-b - Math.Sqrt(delta)) / (2 * a);
            // visualizza soluzioni con tre cifre dopo la virgola
            lblX1.Text = string.Format("{0:f3}", x1);
            lblX2.Text = string.Format("{0:f3}", x2);
        }
    }
```

L'istruzione **if** (a == 0 && b == 0) verifica se a è uguale a zero e (AND) b è uguale a zero: se entrambe le condizioni sono vere (true) allora l' **if** è true e quindi vengono eseguite le due istruzioni contenute all'interno della coppia di {} sottostanti:

```
lblX1.Text = "Equazione impossibile";
lblX2.Text = ""; // svuota lblX2
```

Dopo aver eseguito tali istruzioni verranno saltati i blocchi di istruzioni sottostanti l'**else if** e l'**else**.
Se l'unico coefficiente nullo è a, allora sarà false la condizione all'interno dell' **if** (true && false ha come risultato false), quindi verrà eseguita l'istruzione else if (a == 0) che, dando come risultato true, determinerà l'esecuzione delle due istruzioni seguenti:

```
// visualizza soluzione con tre cifre dopo la virgola
            lblX1.Text = string.Format("{0:f3}", -c / b);
            lblX2.Text = "";
```

Dopo aver eseguito tali istruzioni verrà saltato il blocco d'istruzioni sottostante l'**else**.
Se infine si digitano tre coefficienti a, b, c non nulli, essendo false le condizioni all'interno dell'**if** e dell' l'**else if** verrà eseguito l'ultimo blocco d'istruzioni, quello sottostante l'**else**.
Si noti in particolare l'uso della classe **Math** per il calcolo dell'esponenziale b^2 tramite il metodo Math.Pow(b, 2) e della radice quadrata tramite il metodo Math.Sqrt(delta).
I metodi della classe Math realizzano il calcolo di molte funzioni matematiche.
Anche in questo caso, se uno dei tre TextBox contiene una stringa vuota e si clicca sul pulsante verrà visualizzato un messaggio di errore e l'applicazione verrà chiusa.
Per evitare questo inconveniente modifichiamo il codice nel modo seguente:

```
    private void btnCalcola_Click(object sender, EventArgs e)
    {
        // dichiarazione variabili
        double a, b, c, delta, x1, x2;
        // verifica presenza di una stringa vuota in uno dei tre TextBox
        if (txta.Text == "" || txtb.Text == "" || txtc.Text == "")
            return;
        // acquisisce coefficienti dai TextBox
        a = Convert.ToDouble(txta.Text);
        b = Convert.ToDouble(txtb.Text);
        c = Convert.ToDouble(txtc.Text);
        .
        .
        .
```

```
}
```

L'istruzione **if** (txta.Text == "" || txtb.Text == "" || txtc.Text == "") verifica se il primo TextBox contiene una stringa vuota, oppure (OR) se il secondo TextBox contiene una stringa vuota, oppure (OR) se il terzo TextBox contiene una stringa vuota. E' sufficiente che almeno una delle tre condizioni sia true affinché l' **if** risulti true e venga quindi eseguita l'istruzione **return** che termina l'esecuzione del metodo btnCalcola_Click.
Una soluzione alternativa, che implica l'utilizzo dell'operatore logico negazione (**NOT**), è quella seguente:

```
private void btnCalcola_Click(object sender, EventArgs e)
    {
      // dichiarazione variabili
      double a, b, c, delta, x1, x2;
      // verifica presenza di una stringa vuota in uno dei tre TextBox
      if (!(txta.Text == "" || txtb.Text == "" || txtc.Text == ""))
      {
        // acquisisce coefficienti dai TextBox
        a = Convert.ToDouble(txta.Text);
        b = Convert.ToDouble(txtb.Text);
        c = Convert.ToDouble(txtc.Text);
        // verifica se l'eq. è impossibile
        if (a == 0 && b == 0)
        {
          lblX1.Text = "Equazione impossibile";
          lblX2.Text = ""; // svuota lblX2
        }
        // verifica se l'eq. è di primo grado
        else if (a == 0)
        {
          // visualizza soluzione con tre cifre dopo la virgola
          lblX1.Text = string.Format("{0:f3}", -c / b);
          lblX2.Text = "";
        }
        else //soluzione eq.secondo grado
        {
          // calcola delta = b*b-4ac con metodo Pow(base, esponente) della
          // classe Math per il calcolo dell'esponenziale
          delta = Math.Pow(b, 2) - 4 * a * c;
          // ricava radici equazione con metodo Sqrt() della
          // classe Math per il calcolo della radice quadrata
          x1 = (-b + Math.Sqrt(delta)) / (2 * a);
          x2 = (-b - Math.Sqrt(delta)) / (2 * a);
          // visualizza soluzioni con tre cifre dopo la virgola
          lblX1.Text = string.Format("{0:f3}", x1);
          lblX2.Text = string.Format("{0:f3}", x2);
        }
      }
```

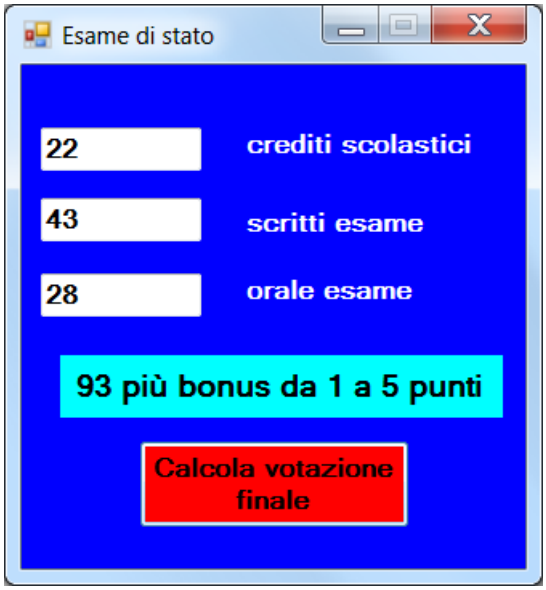
In questo caso l'istruzione **if** (!(txta.Text == "" || txtb.Text == "" || txtc.Text == "")) restituirà true solamente quando le tre condizioni interne saranno tutte false (nessun TextBox contiene una stringa vuota), questo grazie all'operatore logico NOT (**!**) che nega il false ottenendo true; in questo caso saranno eseguite tutte le istruzioni contenute nel blocco di codice sottostante, mentre se almeno un TextBox contiene una stringa vuota, l'istruzione **if** diventerà false (grazie al NOT che nega true facendolo diventare false) e quindi verrà saltata l'esecuzione di tutto il blocco di codice sottostante l'istruzione **if**.

Si noti come le tre condizioni vadano racchiuse all'interno di una coppia di (), questo perché l'operatore NOT deve negare la condizione ottenuta dopo aver applicato gli operatori OR.
Una terza soluzione possibile sarebbe quella di utilizzare al posto dell'operatore logico NOT tre operatori relazionali diverso da (**!=**) modificando l'istruzione if (!(txta.Text == "" || txtb.Text == "" || txtc.Text == "")) nel modo seguente:

```
if (txta.Text != "" &&  txtb.Text  != "" &&  txtc.Text != "")
```

Se la prima TextBox non contiene una stringa vuota e (AND) la seconda TextBox non contiene una stringa vuota e (AND) la terza Textbox non contiene una stringa vuota la condizione risultante all'interno dell' **if** sarà true e quindi sarà eseguito il blocco di codice sottostante.

Come ulteriore esempio di utilizzo dell'istruzione **if** e degli operatori logici vediamo il calcolo del voto finale all'esame di Stato.



Creare un progetto e salvarlo con nome EsameStato.
Al Form vengono aggiunti un controllo Button, tre TextBox - per la digitazione dei crediti scolastici (massimo 25 p.ti), dei tre scritti d'esame (massimo 45 p.ti) e dell'orale (massimo 30 p.ti) - e quattro Label, di cui una dedicata alla visualizzazione del voto finale.
Proprietà Name:
TextBox: txtCredit, txtScritti, txtOrale
Label: lblVoto
Button: btnCalcola

In progettazione doppio click sul pulsante e, nel metodo creato, scrivere il seguente codice:

```
private void btnCalcola_Click(object sender, EventArgs e)
 {
    // dichiarazione delle variabili
    int crediti, scritti, orale, proveEsame, votoFinale;
    // acquisizione dei dati dai TextBox
    crediti = Convert.ToInt32(txtCrediti.Text);
    scritti = Convert.ToInt32(txtScritti.Text);
    orale = Convert.ToInt32(txtOrale.Text);
    // calcola totale prove esame
    proveEsame = scritti + orale;
    // calcola totale esame + crediti
    votoFinale = proveEsame + crediti;
    // visualizzazione risultati
    if (votoFinale == 100)
       lblVotazione.Text = "100 e lode";
    else if(crediti >= 15 && proveEsame >= 70)
       lblVotazione.Text = votoFinale + " più bonus da 1 a 5 punti";
    else if(votoFinale < 60)
       lblVotazione.Text = "Esame non superato";
    else
       lblVotazione.Text = Convert.ToString(votoFinale);
```

```
}
```

Si noti l'utilizzo dell'operatore logico AND (&&) all'interno dell'istruzione else if(crediti >= 15 && proveEsame >= 70): la condizione risulta vera solo se il candidato ha un numero di crediti scolastici maggiore o uguale a 15 e (AND) un totale delle prove d'esame maggiore o uguale a 70.

Istruzione di selezione switch

C# mette a disposizione una seconda istruzione di selezione che consente di effettuare confronti sui molti valori che una variabile può assumere.
Sintassi:

```
switch (variabile)
        {
                case costante1:
                     istruzione1;
                break;
                …….
                case costanteN:
                      istruzioneN;
                break;
                default:        // facoltativa e, se presente, unica
                      istruzione_default;
                break;
          }
```

- Le costanti relative ai **case** devono essere dello stesso tipo della *variabile*
- Se viene trovata una corrispondenza tra il valore della *variabile* e uno dei *case*, vengono eseguite tutte le istruzioni fino alla parola chiave **break che** termina l'esecuzione e "salta" all'istruzione immediatamente successiva al costrutto **switch**
- Se non esiste nessuna corrispondenza, allora il controllo passa alla clausola **default** (se presente) e vengono eseguite le relative istruzioni. Se l'etichetta *default* non è presente e non vi è stata alcuna corrispondenza tra *variabile* ed etichette *case*, allora il controllo passa all'istruzione immediatamente successiva al costrutto **switch**
- L'istruzione **switch** non gestisce tutti i tipi di dati ma solo **tutti i tipi interi** e le **stringhe**.

Come semplice esempio di utilizzo dell'istruzione switch vediamo un programma che richiede la digitazione di un intero tra 1 e 7 e visualizza il giorno della settimana corrispondente.
Creare un progetto e salvarlo con nome GiornoSettimana, quindi aggiungere al Form un TextBox, un Button e due Label.

Proprietà Name:
TextBox: txtNumGiorno
Label: lblGiorno
Button: btnMessaggio

In progettazione doppio click sul pulsante e, nel metodo creato, scrivere il seguente codice:

```
private void btnMessaggio_Click(object sender, EventArgs e)
  {
     // dichiarazione variabile
     byte numGiorno;
     // acquisizione dato da TextBox
     numGiorno = Convert.ToByte(txtNumGiorno.Text);
     switch (numGiorno)
     {
        case 1:  // verifica se la variabile numGiorno contiene la costante 1
           lblGiorno.Text = "E' Lunedì";
           break;
        case 2: // verifica se la variabile numGiorno contiene la costante 2
           lblGiorno.Text = "E' Martedì";
           break;
        case 3: // verifica se la variabile numGiorno contiene la costante 3
           lblGiorno.Text = "E' Mercoledì";
           break;
        case 4:
           lblGiorno.Text = "E' Giovedì";
           break;
        case 5:
           lblGiorno.Text = "E' Venerdì";
           break;
        case 6:
           lblGiorno.Text = "E' Sabato";
           break;
        case 7:
           lblGiorno.Text = "E' Domenica";
           break;
        default:
           lblGiorno.Text = "Nessuna corrispondenza con giorni della settimana";
           break;
     }
  }
```

Se la variabile numGiorno contiene la costante 1 viene eseguita l'istruzione compresa tra **case** 1 e **break**, se invece contiene la costante 2 viene eseguita l'istruzione compresa tra **case** 2 e **break** e così via fino al valore 7; se invece si digita un valore non compreso nell'intervallo 1-7 sarà eseguita l'istruzione compresa tra **default** e **break**. Si osservi l'analogia tra l'istruzione if…else if…else e l'istruzione switch in cui il ruolo dell'else è svolto dal default.

- Può capitare a volte che a un valore della *variabile* di selezione venga associato più di un costrutto *case*; C# prevede questa possibilità tramite la seguente sintassi:

```
switch (variabie)
      {
            case costante1:
            case costante2:
                  istruzione1;
            break;
            …….
            case costante3:
            case costante4:
                  istruzione2;
```

```
            break;
            default:        // facoltativa e, se presente, unica
                istruzione_default;
            break;
        }
```

Vediamo un esempio in cui è richiesta la digitazione di una squadra di calcio tra le otto seguenti (Juventus, Torino, Milan, Inter, Lazio, Roma, Genoa, Sampdoria) e visualizzata la città corrispondente alla squadra.

Proprietà Name:
TextBox: txtSquadra
Label: lblCitta
Button: btnCitta

In progettazione doppio click sul pulsante e, nel metodo creato, scrivere il seguente codice:

```
private void btnCitta_Click(object sender, EventArgs e)
 {
   // svuota Label
    lblCitta.Text = "";
    // dichiarazione variabile
    string squadra;
    // acquisizione dato da TextBox
    squadra = txtSquadra.Text;
    switch (squadra)
    {
       case "Juventus":
       case "Torino":
          lblCitta.Text = "Torino";
          break;
       case "Milan":
       case "Inter":
          lblCitta.Text = "Milano";
          break;
       case "Roma":
       case "Lazio":
          lblCitta.Text = "Roma";
          break;
       case "Genoa":
       case "Sampdoria":
          lblCitta.Text = "Genova";
          break;
    }
```

```
}
```

Se si digita “Juventus” oppure “Torino” verrà eseguita l’istruzione lblCitta.Text = "Torino", analogamente per le altre coppie di squadre della stessa città. In questo costrutto switch non è presente default, quindi se si digita una squadra che non trova corrispondenza in uno dei case, nella Label lblCitta non verrà visualizzato nulla.

Modifichiamo ora il progetto EsameStato aggiungendo al Form un secondo controllo Button per il calcolo dei crediti scolastici, due TextBox per l’inserimento della classe frequentata e della media voti conseguita a fine anno e una Label per la visualizzazione dei punti di credito.

Proprietà Name:
TextBox: txtClasse, txtMediaVoti
Label: lblPuntiCredito
Button: btnCrediti

Analizziamo l’istruzione seguente:

```
classe = classe.ToLower();
```

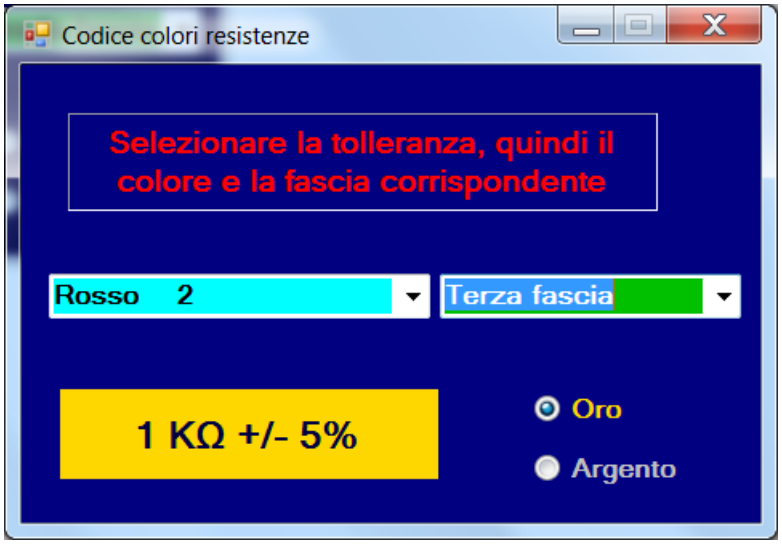
Il metodo Tolower converte in caratteri minuscoli la stringa presente nella variabile classe.
Successivamente l’istruzione **switch** verifica la classe digitata: nel caso in cui sia digitato terza o quarta (in lettre o numeri) il programma, tramite l’istruzione **if…else if…else** annidata all’interno dei primi quattro **case**, calcola i punti di credito scolastico ottenuti. Analogamente, se si digita quinta (in lettere o numero) sarà eseguita l’istruzione **if…else if…else** annidata all’interno degli ultimi due case.
Se si digita come classe una stringa che non ha alcuna corispondenza con i **case**, allora sarà eseguita l’istruzione successiva al **default**.

```
private void btnCrediti_Click(object sender, EventArgs e)
 {
    // dichiarazione delle variabili
    string classe;
    double mediaVoti;
    // acquisizione dei dati dai TextBox
    classe = txtClasse.Text;
    mediaVoti = Convert.ToDouble(txtMediaVoti.Text);
    // converte la stringa nella variabile classe in caratteri minuscoli
    classe = classe.ToLower();
    // verifica classe frequentata
    switch (classe)
    {
       // ricava punti di credito di terza o quarta
       case "terza":
       case "quarta":
       case "3":
       case "4":
          if (mediaVoti == 6)
```

```
            lblPuntiCredito.Text = "3-4 punti di credito";
         else if (mediaVoti > 6 && mediaVoti <= 7)
            lblPuntiCredito.Text = "4-5 punti di credito";
         else if (mediaVoti > 7 && mediaVoti <= 8)
            lblPuntiCredito.Text = "5-6 punti di credito";
         else if (mediaVoti > 8 && mediaVoti <= 9)
            lblPuntiCredito.Text = "6-7 punti di credito";
         else if (mediaVoti > 9)
            lblPuntiCredito.Text = "7-8 punti di credito";
         else
            lblPuntiCredito.Text = "Non promosso";
         break;
      // ricava punti di credito di quinta
      case "quinta":
      case "5":
         if (mediaVoti == 6)
            lblPuntiCredito.Text = "4-5 punti di credito";
         else if (mediaVoti > 6 && mediaVoti <= 7)
            lblPuntiCredito.Text = "5-6 punti di credito";
         else if (mediaVoti > 7 && mediaVoti <= 8)
            lblPuntiCredito.Text = "6-7 punti di credito";
         else if (mediaVoti > 8 && mediaVoti <= 9)
            lblPuntiCredito.Text = "7-8 punti di credito";
         else if (mediaVoti > 9)
            lblPuntiCredito.Text = "8-9 punti di credito";
         else
            lblPuntiCredito.Text = "Non ammesso all'esame di stato";
         break;
      default:
         lblPuntiCredito.Text = "classi consentite: 3, 4, 5";
         break;
   }
}
```

Controlli RadioButton e ComboBox

Vediamo ora un'applicazione che permette di ricavare valore nominale e tolleranza di una resistenza a partire dal suo codice colori (solo per le serie E12 ed E24 con tolleranza, rispettivamente, del 10% e 5%).



Creare un progetto con nome CodiceColoriResistenze, quindi inserire nel Form due Label, due controlli **RadioButton** e due controlli **ComboBox**.
I controlli RadioButton sono molto usati nelle applicazioni Windows in quanto ideali per le situazioni in cui esistono solo due possibilità di scelta, in questo caso tolleranza Oro oppure Argento.
Il controllo ComboBox è una combinazione di una casella di testo e di un elenco a discesa contenente le scelte valide; in questo caso i due ComboBox permettono la selezione della fascia e del relativo colore.

La Label con sfondo giallo permette di visualizzare valore nominale e tolleranza della resistenza.

Proprietà Name:
Label: lblValNom
RadioButton: radOro, radArgento
ComboBox: cmbColoreFascia, cmbNumeroFascia

Impostare la proprietà Text di cmbColoreFascia su "Scegliere il colore" e la proprietà Text di cmbNumeroFascia su "Scegliere la fascia".

- In fase di progettazione fare doppio click in una zona vuota di Form1: verrà creato il metodo Form1_Load().
- Selezionare cmbNumeroFascia e nella finestra delle proprietà cliccare sull'icona a forma di fulmine e selezionare l'evento SelectedValueChanged: verrà creato il metodo cmbNumeroFascia_SelectedValueChanged()
- In fase di progettazione fare doppio click sul RadioButton radArgento: verrà creato il metodo radArgento_CheckedChanged()

```
namespace CodiceColoriResistenze
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
      }
   // variabile globale inizializzata con una stringa vuota
      string valore = "";

      private void Form1_Load(object sender, EventArgs e)
      {
         // metodo eseguito all'avvio del programma, cioè appena si
         // clicca su Play
         // aggiungono a cmbColoreFascia i colori selezionabili
         cmbColoreFascia.Items.Add("Nero      0");
         cmbColoreFascia.Items.Add("Marrone 1");
         cmbColoreFascia.Items.Add("Rosso    2");
         cmbColoreFascia.Items.Add("Arancio  3");
         cmbColoreFascia.Items.Add("Giallo    4");
         cmbColoreFascia.Items.Add("Verde     5");
         cmbColoreFascia.Items.Add("Azzurro  6");
         cmbColoreFascia.Items.Add("Violetto   7");
         cmbColoreFascia.Items.Add("Grigio    8");
         cmbColoreFascia.Items.Add("Bianco    9");
         cmbColoreFascia.Items.Add("Argento   0,01  fascia 3");
         cmbColoreFascia.Items.Add("Oro       0,1   fascia 3");
         // aggiungono a cmbNumeroFascia le fasce selezionabili
         cmbNumeroFascia.Items.Add("Prima fascia");
         cmbNumeroFascia.Items.Add("Seconda fascia");
         cmbNumeroFascia.Items.Add("Terza fascia");
         // seleziona radArgento
         radArgento.Checked = true;
         // imposta colore di sfondo di lblValNom
```

```
    lblValNom.BackColor = Color.Silver;
}

// metodo eseguito quando cambia la fascia selezionata all'interno di cmbNumeroFascia
// verifica la fascia selezionata convertendo in stringa l'elemento selezionato
private void cmbNumeroFascia_SelectedValueChanged(object sender, EventArgs e)
{
    double valNom, pot;
    int valNorm, molt;
    string valoreNomin;// variabile con valore nominale ed unità di misura

    switch (cmbNumeroFascia.SelectedItem.ToString())
    {
        // la proprietà mbColoreFascia.SelectedIndex vale 0 per il colore Nero, 1 per il Marrone
        // e così via
        case "Prima fascia":
            valore = cmbColoreFascia.SelectedIndex.ToString();
            break;
        case "Seconda fascia":
            valore = valore + cmbColoreFascia.SelectedIndex.ToString();// concatena i valori delle prime
                //due fasce
            break;
        case "Terza fascia":
            // se valore contiene stringa vuota termina metodo cmbNumeroFascia_SelectedValueChanged()
            if (valore == "")
                return;
            // ricava valore normalizzato
            valNorm = Convert.ToInt32(valore);
            // ricava potenza di 10
             molt = cmbColoreFascia.SelectedIndex;
            if (molt == 10)
                pot = 0.01;
            else if (molt == 11)
                pot = 0.1;
            else
                pot = Math.Pow(10, molt);
            // ricava valore nominale
            valNom = valNorm * pot;
          // converte valore nominale in KΩ, MΩ oppure Ω
            if (valNom / 1000 >= 1 && valNom / 1000 < 1000)
                valoreNomin = (valNom / 1000.0).ToString() + " KΩ";
            else if(valNom / 1000000 >= 1)
                valoreNomin = (valNom / 1000000.0).ToString() + " MΩ";
            else
                valoreNomin = valNom.ToString() + " Ω";
            // visualizza valore nominale e tolleranza
            if (radOro.Checked)
                lblValNom.Text = valoreNomin + " +/- 5%";
            else
                lblValNom.Text = valoreNomin + " +/- 10%";
            break;
    }
}
```

```
        private void radArgento_CheckedChanged(object sender, EventArgs e)
        {
            // imposta colore di sfondo di lblValNom
            if (radArgento.Checked)
                lblValNom.BackColor = Color.Silver;
            else
                lblValNom.BackColor = Color.Gold;
            // svuota Label
            lblValNom.Text = "";
        }
    }
}
```

Come prima cosa si osservi la dichiarazione della variabile *valore*: tale dichiarazione risulta all'interno della classe **Form1**, ma al di fuori di tutti i metodi che in essa sono contenuti, in particolare al di fuori dei tre metodi Form1_Load(), cmbNumeroFascia_SelectedValueChanged() e radArgento_CheckedChanged() in cui scriviamo il codice del programma. La variabile *valore* viene quindi detta globale, in quanto è possibile utilizzarla in tutti i metodi creati all'interno della classe **Form1**.
Le variabili dichiarate all'interno di un metodo, ad esempio le variabili dichiarate nel metodo cmbNumeroFascia_SelectedValueChanged(), sono dette *locali* in quanto il loro uso consentito solo all'interno di tale metodo: se ad esempio si prova ad usare la variabile *valNom* - dichiarata in cmbNumeroFascia_SelectedValueChanged() - nel metodo Form1_Load() viene generato un messaggio di errore durante la compilazione.
Il metodo Form1_Load() viene eseguito quando si verifica l'evento Load sul Form1, cioè non appena si esegue il programma cliccando su menu Debug | Avvia senza eseguire debug.
L'istruzione cmbColoreFascia.Items.Add("Nero 0") aggiunge come prima opzione nell'elenco a discesa del ComboBox cmbColoreFascia la stringa "Nero 0", analogamente le istruzioni successive aggiungono in successione le altre opzioni per la scelta del colore di una fascia.
Vengono poi aggiunte al ComboBox cmbNumeroFascia le tre opzioni relative al numero della fascia.
L'istruzione seguente:

```
radArgento.Checked = true;
```

assegna alla proprietà Checked del controllo radArgento il valore booleano **true**: ciò fa si che, all'avvio del programma, risulti selezionato il RadioButton in questione e deselezionato il RadioButton radOro.
Si noti che tutti i RadioButton, qualsiasi sia il loro numero, sono in relazione tra di loro e può esserne selezionato solo uno per volta.
L'istruzione lblValNom.BackColor = **Color**.Silver assegna alla proprietà BackColor (colore di sfondo) del controllo lblValNom il colore Silver (argento).
Il metodo radArgento_CheckedChanged() viene eseguito ogni volta che cambia lo stato di selezione del RadioButton radArgento, cioè sia che si selezioni il RadioButton in questione, sia che lo si deselezioni cliccando sul RadioButton radOro.
All'interno del metodo l'istruzione **if…else** testa la proprietà Checked del RadioButton radArgento: e la proprietà è **true** viene eseguita l'istruzione lblValNom.BackColor = **Color**.Silver che imposta come Argento il colore di sfondo della Label lblValNom, mentre se tale proprietà è **false** viene eseguita l'istruzione lblValNom.BackColor = **Color**.Gold che imposta come Oro il colore di sfondo della Label.
Analizziamo ora il codice presente all'interno del metodo cmbNumeroFascia_SelectedValueChanged(), metodo eseguito ogni volta che si verifica sul controllo ComboBox cmbNumeroFascia l'evento SelectedValueChanged, cioè ogni volta che cambia la fascia selezionata all'interno di cmbNumeroFascia.
Con l'istruzione:

```
switch (cmbNumeroFascia.SelectedItem.ToString())
```

si verifica l'elemento (la fascia) selezionato nel ComboBox cmbNumeroFascia, dopo averlo convertito in una stringa: se si trova una corrispondenza nel **case** "Prima fascia" viene eseguita l'istruzione:

```
valore = cmbColoreFascia.SelectedIndex.ToString();
```

L'istruzione precedente converte l'indice numerico dell'elemento selezionato (SelectedIndex) nel ComboBox cmbColoreFascia in una stringa salvandola nella variabile valore; si noti che nei controlli ComboBox il primo elemento dell'elenco a discesa ha come indice numerico zero, il secondo uno, il terzo due e così via; nel nostro caso il colore nero avrà indice zero, il marrone uno, il rosso due e così via.
Se l'istruzione **switch** trova invece una corrispondenza in **case** "Seconda fascia" viene eseguita l'istruzione:

```
valore = valore + cmbColoreFascia.SelectedIndex.ToString();
```

L'istruzione precedente converte l'indice numerico dell'elemento selezionato nel ComboBox cmbColoreFascia in una stringa e la concatena con il contenuto della variabile valore.
Supponiamo, ad esempio, di aver selezionato il marrone come prima fascia ed il nero come seconda fascia: allora la variabile *valore* conterrà inizialmente la stringa "1" che, concatenata alla stringa "0" darà origine alla stringa "10" che sarà memorizzata nella variabile *valore* sovrascrivendone il valore precedente.
Se infine si seleziona la terza fascia, e viene quindi trovata una corrispondenza con **case** " Terza fascia", allora l'istruzione seguente salva nella variabile *valNorm* il valore normalizzato convertendo in un numero intero la stringa presente nella variabile *valore* (ad es. converte la stringa "10" nel numero 10):

```
valNorm = Convert.ToInt32(valore);
```

Le istruzioni successive ricavano la potenza di dieci per cui moltiplicare il valore normalizzato:

```
molt = cmbColoreFascia.SelectedIndex;
        if (molt == 10)
           pot = 0.01;
        else if (molt == 11)
           pot = 0.1;
        else
           pot = Math.Pow(10, molt);
```

La variabile *molt* memorizza l'indice numerico dell'elemento selezionato nel ComboBox cmbColoreFascia: se l'indice è pari a dieci significa che si è selezionato l'undicesimo colore, cioè l'Argento, corrispondente ad un moltiplicatore pari a 0.01; se invece l'indice è pari a undici significa che si è selezionato il dodicesimo colore, cioè l'Oro, corrispondente ad un moltiplicatore pari a 0.1. Per tutti gli altri colori, il valore del moltiplicatore si ottiene elevando dieci ad un esponente pari all'indice numerico del colore selezionato (ad es., se si è selezionato il Rosso – terzo colore, quindi molt = 2 – il moltiplicatore sarà 10^2.
Per ottenere il valore nominale è sufficiente moltiplicare il valore normalizzato (*valNorm*) per il moltiplicatore(*pot*).
Le istruzioni rimanenti convertono il valore nominale in KΩ, MΩ oppure Ω e ne visualizzano il valore unitamente alla tolleranza.

Istruzione iterativa while

Sintassi:

```
while(condizione)
        istruzione_true;
```

E' possibile sostituire a istruzione_true un ***blocco di codice*** composto da due o più istruzioni racchiuse tra **{ }**.

- L'istruzione **while** valuta la *condizione* e, se risulta vera(*true*), è eseguita istruzione_true. Una volta eseguita l'istruzione, viene nuovamente valutata la *condizione* e, solo se risulta falsa(*false*), il flusso del programma prosegue con l'istruzione successiva al costrutto while, altrimenti viene nuovamente eseguita istruzione_true e così via.

Come primo esempio di utilizzo dell'istruzione **while** vediamo la conversione decimale-binario di un numero intero privo disegno.

Creare un progetto con nome ConversioneDecimaleBinario, quindi aggiungere al Form un Button, un TextBox ed una Label.

Proprietà Name:
TextBox: txtNumero
Label: lblRisultato
Button: btnConverti

In progettazione doppio click sul pulsante e, nel metodo creato, scrivere il seguente codice:

```
private void btnConverti_Click(object sender, EventArgs e)
  {
    // dichiarazione variabili
    uint decimale = 0, bit;
    string binario = ""; // stringa vuota
      // acquisizione numero decimale
      decimale = Convert.ToUInt32(txtNumero.Text);
      // conversione decimale-binario
      while (decimale != 0)
      {
        bit = decimale % 2;  // ricava resto (bit)
        decimale = decimale / 2; // ricava il quoziente
       // concatena a sinistra ultimo bit ottenuto ai bit precedenti
        binario = bit + binario;
      }
   // visualizzazione numero binario
    lblRisultato.Text = binario;
  }
```

L'istruzione **while** verifica subito se la variabile *decimale* contiene un numero diverso da zero e, se tale condizione è vera viene eseguito il codice presente all'interno delle {} sottostanti il **while**.
Come prima cosa ottiene il primo resto (il bit meno significativo) dividendo *decimale* per due mediante l'operatore resto (%), successivamente aggiorna il quoziente dividendo *decimale* per due mediante l'operatore quoziente (/), quindi concatena a sinistra il primo resto alla stringa vuota.
Se ad esempio, la variabile *decimale* contiene inizialmente il numero cinque, il primo resto sarà 1 e il primo quoziente due.
Concatenando 1 con una stringa vuota si otterrà una stringa formata solamente dal carattere 1, memorizzata al posto della stringa vuota nella variabile *binario*. A questo punto, viene nuovamente valutata la condizione nel **while**, che sarà ancora true poiché la variabile *decimale* contiene un quoziente pari a due: vengono quindi nuovamente eseguite le istruzioni tra {} che forniranno un nuovo resto pari a zero e un nuovo quoziente pari a uno; concatenando il carattere zero con la stringa contenuta nella variabile *binario* – contenente il carattere 1 – si otterrà la stringa "01".

Viene nuovamente valutata la condizione nel **while**, che sarà ancora true poiché la variabile *decimale* contiene un quoziente pari a uno: vengono quindi nuovamente eseguite le istruzioni tra {} che forniranno un nuovo resto pari a uno e un nuovo quoziente pari a zero; concatenando il carattere 1 con la stringa "01" contenuta nella variabile *binario* si otterrà la stringa "101" che è la rappresentazione binaria del numero cinque decimale.
Viene infine nuovamente valutata la condizione nel **while**, che questa volta sarà false poiché la variabile *decimale* contiene un quoziente pari a zero: il ciclo **while** termina e quindi l'esecuzione del codice prosegue con l'istruzione successiva costrutto while, cioè con l'istruzione lblRisultato.Text = binario che visualizza nella Label il contenuto della variabile *binario*, cioè la rappresentazione binaria del numero decimale digitato.

Istruzione iterativa do...while

Sintassi:

do
istruzione_true;
while(*condizione*);

E' possibile sostituire a istruzione_true un ***blocco di codice*** composto da due o più istruzioni racchiuse tra **{ }**.

- A differenza dell'istruzione while, l'istruzione **do…while** fa si che l'istruzione_true venga eseguita almeno una volta.

Come esempio di applicazione dell'istruzione **do…while** proviamo a modificare l'applicazione precedente sotituendo tale istruzione all'istruzione **while**. Il codice diventa il seguente:

```
private void btnConverti_Click(object sender, EventArgs e)
 {
    // dichiarazione variabili
    uint decimale = 0, bit;
    string binario = ""; // stringa vuota
       // acquisizione numero decimale
       decimale = Convert.ToUInt32(txtNumero.Text);
       // conversione decimale-binario
       do
       {
          bit = decimale % 2;
          decimale = decimale / 2;
          binario = bit + binario;
       }
       while (decimale != 0) ;
   // visualizzazione numero binario
    lblRisultato.Text = binario;
 }
```

Con la soluzione vista in precedenza, se si digita il numero decimale zero, nella Label non viene visualizzato nulla, questo perché la condizione nel **while** risulta subito falsa e quindi le istruzioni al suo interno non sono eseguite neppure una volta.
Con l'utilizzo dell'istruzione **do…while**, invece, prima vengono eseguite le istruzioni tra {} e poi verificata la condizione nel **while**; ciò fa si che le istruzioni tra {} vengano eseguite una volta visualizzando il numero zero nella Label.

Istruzione iterativa for

Le istruzioni iterative precedenti sono utili quando non si conosce a priori quante volte devono essere eseguite le istruzioni da iterare (ripetere). Se invece si conosce tale numero è meglio usare la struttura for.

Sintassi:

for*(inizializzazione; condizione; incremento)*
istruzione;

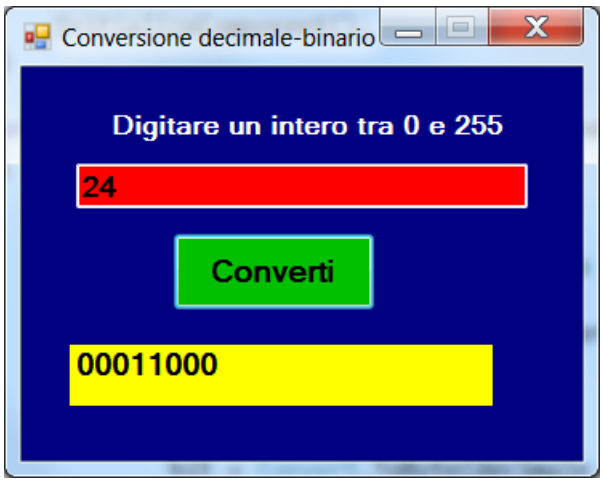
E' possibile sostituire a istruzione un ***blocco di codice*** composto da due o più istruzioni racchiuse tra **{ }**.

- *inizializzazione* rappresenta l'espressione in cui è dichiarato e inizializzato un contatore (es. byte i = 1;)
- *condizione* rappresenta l'espressione in cui compare la condizione di controllo (es. i <= 5)
- *incremento* rappresenta l'espressione in cui avviene l'incremento del contatore (es. i++)

- Inizialmente viene eseguita (una sola volta) l'*inizializzazione*, poi controllata la *condizione* e, se questa risulta vera, sono elaborate tutte le istruzioni all'interno del blocco di codice, altrimenti sono eseguite le istruzioni successive al ciclo. Una volta elaborate le istruzioni interne al ciclo viene eseguito l'*incremento* e poi nuovamente verificata la *condizione* e così via.

- A volte può essere necessario interrompere un ciclo ***for*** (oppure ***while*** o ***do…while***) prematuramente oppure passare immediatamente all'iterazione successiva senza dover obbligatoriamente completare l'intero ciclo. Le due istruzioni in questione sono, rispettivamente, **break** e **continue**: la prima consente di "saltare" alla prima istruzione utile dopo il ciclo ***for*** (oppure ***while*** o ***do…while***), mentre la seconda consente di "saltare" direttamente al test condizionale di controllo del ciclo, evitando l'elaborazione dell'eventuale codice successivo. In particolare l'uso di **continue** in un ciclo *for* determina l'incremento del contatore e quindi la successiva verifica della *condizione* di controllo,

- Le variabili dichiarate nell'espressione di inizializzazione di un ciclo **for** restano visibili solo all'interno del ciclo per tutta la durata del medesimo. Una volta terminate tutte le iterazioni, tali variabili non sono più disponibili, cioè non sono più accessibili da parte del codice successivo a meno che non le si dichiari nuovamente.



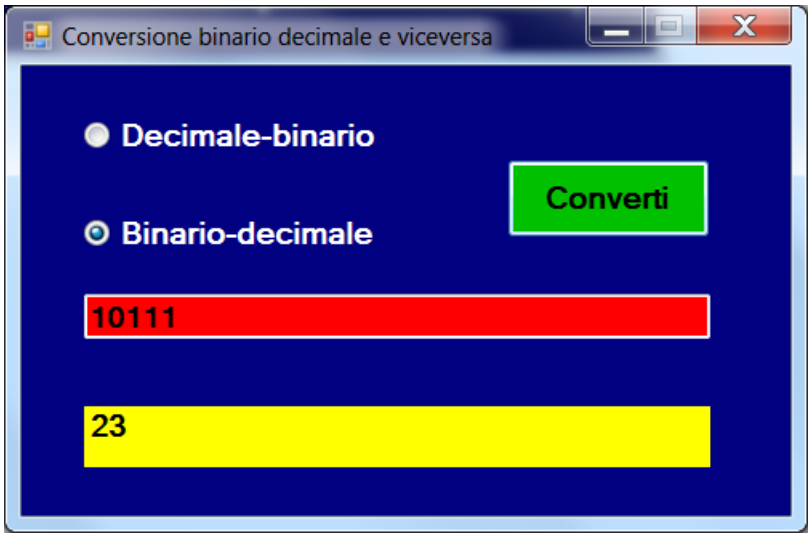
Utilizziamo ora l'istruzione ***for*** per convertire in binario un numero tra 0 e 255 visualizzando però sempre otto bit; per realizzare ciò è necessario eeguire sempre otto divisioni – otto resti e otto quozienti – indipendentemente dal numero digitato.

Modifichiamo quindi il codice del progetto precedente sostituendo il ciclo **while** con quello **for**:

```
private void btnConverti_Click(object sender, EventArgs e)
{
   // dichiarazione variabili
   byte decimale = 0, bit;
   string binario = ""; // stringa vuota
      // acquisizione numero decimale
      decimale = Convert.ToByte(txtNumero.Text);
```

```
        // conversione decimale-binario
        for (int i = 1; i <= 8; i++)
        {
            bit = Convert.ToByte(decimale % 2);
            decimale = Convert.ToByte(decimale / 2);
            binario = bit + binario;
        }
      // visualizzazione numero binario
       lblRisultato.Text = binario;
}
```

Analizziamo il funzionamento del ciclo **for**. Come prima istruzione viene eseguita la dichiarazione della variabile contatore ***i*** e la sua inizializzazione ad uno, quindi viene valutata la condizione i <= 8; essendo tale condizione vera (true) vengono eseguite le tre istruzioni tra le {} e, dopo aver eseguito l'ultima, viene incrementato il contenuto della variabile ***i***, che diventa pari a due, e poi nuovamente valutata la condizione i <= 8. La condizione è ancora vera e quindi viene ancora eseguito il blocco di istruzioni all'interno del ciclo **for** ed incrementata la variabile ***i*** che assume il valore tre. L'esecuzione del ciclo **for** continua per tutti i valori della variabile contatore ***i*** minori o uguali a otto; in particolare dopo aver eseguito per l'ottava volta il ciclo **for,** la variabile sarà ancora incrementata e il suo valore portato a nove: ora la condizione diveterà falsa (nove non è minore o uguale a otto), quindi termina il ciclo **for** l'esecuzione de programma passa all'istruzione successiva al costrutto **for,** cioè alla visualizzazione del numero binario.
Si noti la somiglianza del costrutto **for** con il costrutto **while**: entrambe le istruzioni, a differenza dell'istruzione **do…while,** prima valutano la condizione e poi, eventualmente, eseguono le relative istruzioni.
Nel **while** e nel **for**, quindi, può succedere che le relative istruzioni non siano mai eseguite (se la condizione di controllo è falsa già la prima volta che viene eseguita), mentre l'istruzione **do…while** garantisce che le relative istruzioni siano eseguite almeno una volta.
Applichiamo ora il ciclo **for** alla conversione binario-decimale.



Riapriamo il progetto ConversioneDecimaleBinario e modifichiamolo aggiungendo al Form due RadioButton con proprietà Name radDecBin e radBinDec.

Creiamo il metodo Form1_Load() e inseriamo l'istruzione che selezioni, all'avvio del programma, la conversione decimale-binario, quindi modifichiamo il metodo btnConverti_Click().

```
private void btnConverti_Click(object sender, EventArgs e)
    {
       // dichiarazione variabile intera a 32 bit priva di segno
       uint decimale = 0, bit;
      // variabile string inizializzata con stringa vuota
       string binario = "";
       // verifica se radDecBin è selezionato
       if (radDecBin.Checked)
       {
```

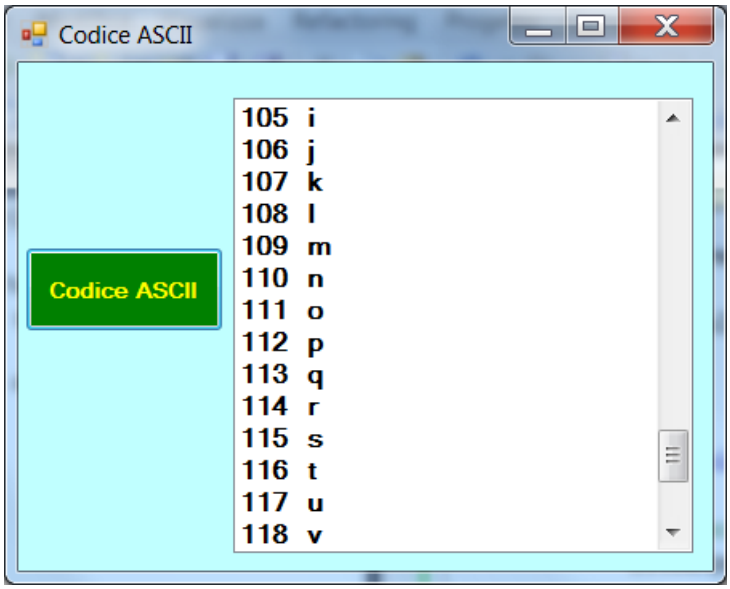
```
        // acquisizione numero decimale
        decimale = Convert.ToUInt32(txtNumero.Text);
        // conversione decimale-binario
        while (decimale != 0)
        {
           bit = decimale % 2;
           decimale = decimale / 2;
           binario = bit + binario;
        }
        lblRisultato.Text = binario;
     }
     else
  {
        // conversione binario-decimale
        // acquisisce stringa con numero binario
        binario = txtNumero.Text;
        // ricava numero di bit della stringa binario
        int lunghezza = binario.Length;
        // ricava posizione lsb
        int posLsb = lunghezza - 1;
        for (int i = posLsb; i >= 0; i--)
        {
           // preleva i-esimo bit dalla stringa binaria
           bit = Convert.ToUInt32(binario.Substring(i, 1));
           if (bit == 1)
              decimale = Convert.ToUInt32(decimale + Math.Pow(2, posLsb - i));
        }
        lblRisultato.Text = decimale.ToString();
     }
  }

  private void Form1_Load(object sender, EventArgs e)
  {
     // seleziona il RadioButton per la conversione decimale-binario
     radDecBin.Checked = true;
  }
```

Analizzando il codice si osserva che l'istruzione if (radDecBin.Checked) risulta vera se ad essere selezionato (proprietà Checked true) è il RadioButton radDecBin; in questo caso vengono eseguite le istruzioni per la conversione decimale-binario, mentre risulta falsa se ad essere selezionato (proprietà radDecBin.Checked false) è l'altro RadioButton di nome radBinDec. In quest'ultimo caso verranno eseguite le istruzioni sotto l'**else**, istruzioni che realizzano la conversione binario-decimale.
Esaminiamo quindi le istruzioni all'interno del blocco **else.**
La variabile *lunghezza* viene inizializzata da un intero che rappresenta la lunghezza della stringa memorizzata nella variabile *binario*; dove per lunghezza di una stringa si intende il numero di caratteri che la compongono.
Si tenga presente che i caratteri che compongono una stringa sono numerati da sinistra a destra partendo da zero; ad esempio nella stringa "pane" la 'p' è il carattere numero zero, la 'a' il carattere numero uno, la 'n' il numero due e la 'e' il carattere numero tre. Possiamo anche osservare che la stringa "pane" ha una lunghezza pari a quattro che, se diminuita di uno, fornisce esattamente la posizione dell'ultimo carattere a destra (la lettera e).
Nel nostro caso, essendo la striga memorizzata nella variabile *binario* la rappresentazione di un numero binario formata dai bit/caratteri '0' e '1', la variabile *lunghezza* sarà inizializzata con il numero di bit di tale

stringa, mentre la variabile *posLsb* memorizzerà la posizione dell'ultimo carattere della stringa in questione, cioè la posizione del bit meno significativo (Lsb).
La variabile contatore ***i*** del ciclo **for** sarà quindi inizializzata con la posizione dell'Lsb e decrementata di uno (i--) ad ogni loop fino ad arrivare al bit più significativo, quello con posizione i = 0.
L'istruzione:

```
bit = Convert.ToUInt32(binario.Substring(i, 1));
```

estrae dalla variabile *binario* una sottostringa formata da un carattere, partendo dal carattere in i-esima posizione; in pratica estrae un bit alla volta - sotto forma di carattere '0' e '1'- partendo dal meno significativo, convertendolo poi nell' intero corrispondente.
Successivamente, se il bit vale uno, occorre ricavare la potenza di due appropriata tramite l'istruzione Math.Pow(2, posLsb - i)).
Se ad es. si è digitato un numero a quattro bit, la variabile *posLsb* conterrà il numero tre e quindi quando ***i*** vale tre (Lsb) il metodo Math.Pow() restituirà $2^{3-3} = 2^0 = 1$, che è la potenza di due abbinata all'Lsb.
Analogamente, il penultimo bit (i=2) darà origine alla potenza $2^{3-2} = 2^1 = 2$ e così via fino ad arrivare al bit più significativo (i=0) che origine alla potenza $2^{3-0} = 2^3 = 8$.
Ogni volta che un bit vale uno, la relativa potenza viene sommata con quelle precedenti salvando il risultato nella variabile *decimale*; al termine del ciclo for quest'ultima conterrà il numero decimale corrispondente al binario digitato.
Vediamo ora un ulteriore esempio di utilizzo del ciclo **for** per la stampa in un controllo **ListBox** dei caratteri con codice ASCII compreso tra 32 e 126.



Creare un nuovo progetto e salvarlo con nome CodiceAscii.
Aggiungere al Form un controllo Button (btnAscii) e un controllo ListBox (lstAscii).
Il controllo **ListBox** è simile al controllo ComboBox e permette di visualizzare tutti i caratteri il cui codice ASCII è compreso tra 32 e 126.
La barra di scorrimento verticale del ListBox viene automaticamente aggiunta quando risulta necessaria per visualizzare tuti gli elementi presenti all'interno del ListBox.
Esaminiamo il codice presente all'interno del metodo btnAscii_Click().
La prima istruzione cancella gli eventuali elementi presenti nel ListBox.

Il ciclo **for** genera tutti i numeri interi compresi tra 32 e 126, mentre la prima istruzione al suo interno costruisce, mediante il metodo Format(), una stringa composta dal codice ASCII e dal relativo carattere, separati da degli spazi vuoti.
La stringa ottenuta viene poi aggiunta al ListBox.

```
private void btnAscii_Click(object sender, EventArgs e)
    {
        // cancella il contenuto del ListBox
        lstAscii.Items.Clear();
        // variabile contenente carattere e relativo codice ASCII
        string riga;
        for(int i = 32; i <= 126; i++)
        {
          // costruisce stringa contenente codice ASCII e carattere corrispondente
          riga = string.Format("{0}  {1}", i, Convert.ToChar(i));
```

```
        // aggiunge al ListBox la stringa contenente codice ASCII e carattere corrispondente
        lstAscii.Items.Add(riga);
    }
```

Matrici C#

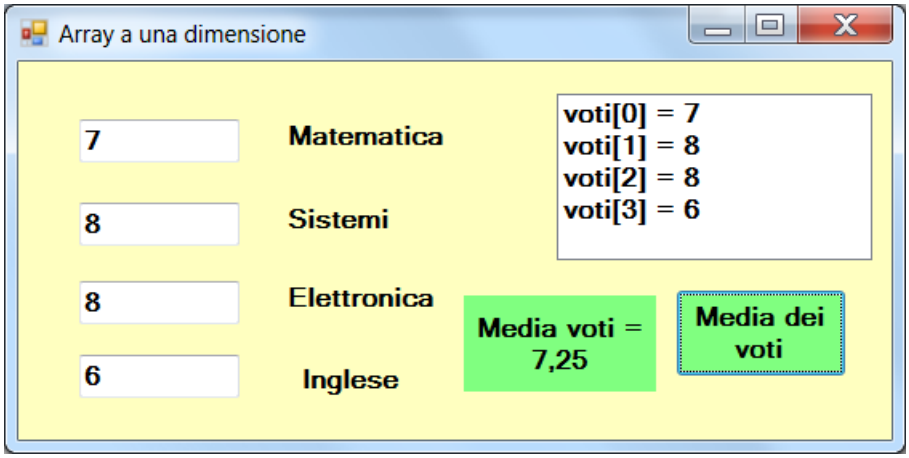
Le strutture che contengono dati tutti dello stesso tipo prendono il nome di ***array*** (matrici) e possono essere matrici a una dimensione (dette anche **vettori**) o matrici a due o più dimensioni.
L'istruzione per **definire una matrice a una dimensione** ha il seguente formato:

tipo [] NomeVettore = new tipo [dimensione]

dove:

- **NomeVettore** è il nome collettivo dei componenti del vettore
- **dimensione** è il numero degli elementi del vettore
- **tipo** specifica il tipo di dato comune a tutte le componenti
- **new** è la parola chiave che indica che viene definita una nuova struttura di dati

Ogni elemento della matrice è individuato da un ***indice*** numerico e l'intervallo dei suoi possibili valori parte da **0** e arriva fino a **dimensione – 1.**
L'indice va racchiuso tra parentesi quadre dopo il nome della variabile.
Come primo esempio vediamo un'applicazione Windows che consenta la digitazione dei voti di quattro materie, salvi tali voti in una matrice ad una dimensione, ne calcoli la media e visualizzi in un Listbox i voti memorizzati nell'array.



Creare un progetto di nome **MatriciVoti**.
Aggiungere al Form un controllo Button, cinque Label, quattro TextBox e un ListBox.
Proprietà Name:
Label: lblMedia
Button: btnMedia
TextBox: txtMatematica, txtSistemi, txtElettronica, txtInglese
ListBox: lstVoti

In progettazione doppio click sul pulsante e, nel metodo creato, scrivere il seguente codice:

```
private void btnMedia_Click(object sender, EventArgs e)
    {
        double mediaVoti = 0;
        // dichiarazione matrice da 4 elementi interi
        int [] voti = new int[4];
        // acquisizione dati e caricamento matrice
        voti[0] = Convert.ToInt32(txtMatematica.Text);
        voti[1] = Convert.ToInt32(txtSistemi.Text);
        voti[2] = Convert.ToInt32(txtElettronica.Text);
        voti[3] = Convert.ToInt32(txtInglese.Text);
        // calcolo somma dei voti
        for(int i = 0; i <= 3; i++)
            mediaVoti = mediaVoti + voti[i];
```

```
        // calcola media dei voti
        mediaVoti = mediaVoti / 4;
        // visualizza media dei voti
        lblMedia.Text = string.Format("Media voti = {0:f2}",mediaVoti);
        // visualizza nel ListBox i voti memorizzati nella matrice
        for(int i = 0; i <= 3; i++)
            lstVoti.Items.Add("voti[" + i + "] = " + voti[i]);
    }
```

L'istruzione:

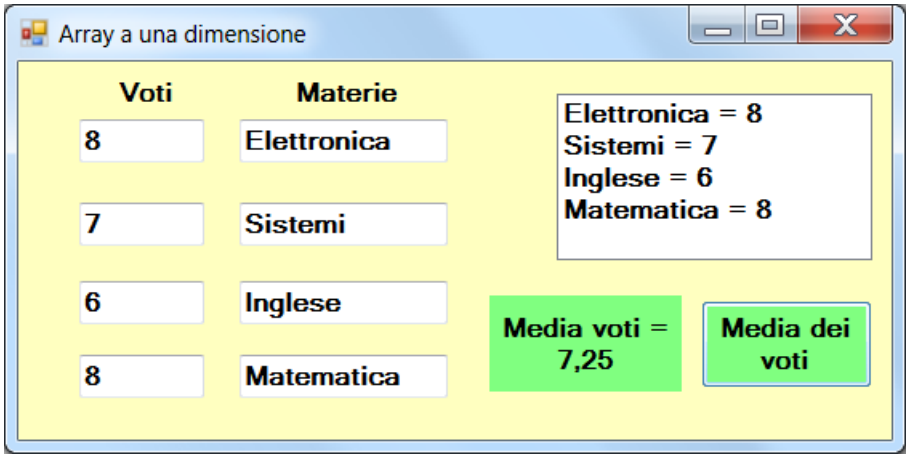
```
voti[0] = Convert.ToInt32(txtMatematica.Text);
```

memorizza il numero intero digitato nel TextBox con proprietà name txtMatematica nel primo elemento della matrice ***voti***, elemento individuato dall'indice numerico zero.
Analogamente gli altri voti saranno memorizzati nel secondo elemento della matrice (indice pari a uno), nel terzo (indice pari a due) e nel quarto di indice pari a tre.
La matrice ***voti*** può essere vista come un vettore formato da quattro righe, numerate da 0 a 3.
La somma dei quattro interi memorizzati nella matrice ***voti*** è realizzata usando un ciclo **for** all'interno del quale la variabile contatore ***i*** conterrà gli indici da 0 a 4 che permettono di individuare i quattro elementi da sommare.
Allo stesso modo un secondo ciclo **for** consente la visualizzazione all'interno del Listbox dei quattro voti memorizzati nell'array di nome ***voti***.

Nell'esempio seguente viene riportato il codice di un'***Applicazione console*** che permette il caricamento e la stampa di una matrice di cinque elementi.

```
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication5
{
    class Program
    {
        static void Main(string[] args)
        {
            // definisce la matrice di 5 elementi
            int[] vett = new int[5];
            for (int i = 0; i < 5; i++)
            {
                // richiede un dato
                Console.Write("Inserisci un dato => ");
                vett[i] = Convert.ToInt32(Console.ReadLine());
            }
            Console.WriteLine("\nContenuto del vettore: ");
            // Stampa gli elementi del vettore
            for (int i = 0; i < 5; i++)
                Console.WriteLine(" vett{0} = {1}", i, vett[i]);
        }
    }
}
```

Modifichiamo ora il progetto MatriciVoti in modo che l'applicazione consenta, oltre alla digitazionedei voti, anche la digitazione dei nomi delle quattro materie.
Rinominare i quattro TextBox nel modo seguente: txtVoto1, txtVoto2, txtVoto3, txtVoto4



Aggiungere quindi altri quattro TextBox con le seguenti proprietà Name:
txtMateria1, txtMateria2, txtMateria3, txtMateria4

Modificare infine il codice all'interno del metodo btnMedia_Click() nel modo seguente:

```
private void btnMedia_Click(object sender, EventArgs e)
    {
      double mediaVoti = 0;
      // dichiarazione matrice da 4 elementi interi
      int [] voti = new int[4];
      // dichiarazione matrice da 4 elementi string
      string [] materie = new string[4];
      // acquisizione voti e materie e caricamento matrici
      voti[0] = Convert.ToInt32(txtVoto1.Text);
      voti[1] = Convert.ToInt32(txtVoto2.Text);
      voti[2] = Convert.ToInt32(txtVoto3.Text);
      voti[3] = Convert.ToInt32(txtVoto4.Text);
      materie[0] = txtMateria1.Text;
      materie[1] = txtMateria2.Text;
      materie[2] = txtMateria3.Text;
      materie[3] = txtMateria4.Text;
      // calcolo somma dei voti
      for(int i = 0; i <= 3; i++)
         mediaVoti = mediaVoti + voti[i];
      // calcola media dei voti
      mediaVoti = mediaVoti / 4;
      // visualizza media dei voti
      lblMedia.Text = string.Format("Media voti = {0:f2}",mediaVoti);
      // visualizza nel ListBox voti e materie memorizzati nelle matrici
      for(int i = 0; i <= 3; i++)
         lstVoti.Items.Add(materie[i] + " = " + voti[i]);
    }
```

Vediamo ora le ***matrici a due dimensioni***.
Le matrici bidimensionali possono essere pensate come tabelle formate da righe e colonne, quindi occorre definire due dimensioni rappresentanti, rispettivamente, il numero di righe e il numero di colonne.
Saranno quindi necessari due indici numerici, il primo rappresentante il numero di riga, il secondo il numero di colonna.
Come esempio, proviamo a modificare il progetto precedente sostituendo alle due matrici ***voti*** e ***materie*** (rispettivamente di tipo **int** e **string**) un'unica matrice a due dimensioni di tipo **string**.
La matrice sarà composta da due colonne (una colonna per i voti, l'altra per le materie) e da quattro righe (quattro materie).
Modifichiamo quindi il codice all'interno del metodo btnMedia_Click() nel modo seguente:

```
private void btnMedia_Click(object sender, EventArgs e)
  {
     double mediaVoti = 0;
     // dichiarazione matrice bidimensionale da 4 righe e due colonne di tipo string
     string [,] votiMaterie = new string[4,2];
     // acquisizione voti e materie e caricamento matrici
     votiMaterie[0,0] = txtVoto1.Text;
     votiMaterie[1,0] = txtVoto2.Text;
     votiMaterie[2,0] = txtVoto3.Text;
     votiMaterie[3,0] = txtVoto4.Text;
     votiMaterie[0,1] = txtMateria1.Text;
     votiMaterie[1,1] = txtMateria2.Text;
     votiMaterie[2,1] = txtMateria3.Text;
     votiMaterie[3,1] = txtMateria4.Text;
     // calcolo somma dei voti
     for(int i = 0; i <= 3; i++)
        mediaVoti = mediaVoti + Convert.ToDouble(votiMaterie[i,0]);
     // calcola media dei voti
     mediaVoti = mediaVoti / 4;
     // visualizza media dei voti
     lblMedia.Text = string.Format("Media voti = {0:f2}",mediaVoti);
     // visualizza nel ListBox voti e materie memorizzati nelle matrice bidimensionale
     for (int i = 0; i <= 3; i++)
        lstVoti.Items.Add(votiMaterie[i,1] + " = " + votiMaterie[i,0]);
  }
```

Analizzando l'istruzione di dichiarazione della matrice a due dimensioni ***votiMaterie***, si nota la presenza di una virgola all'interno delle [], virgola che separa anche il numero delle righe (quattro) dal numero di colonne (due).
Nelle istruzioni di acquisizione e di caricamento della matrice si osserva l'utilizzo di due indici all'interno delle []: il primo individua il numero della riga, il secondo il numero di colonna.
Si osservi come la prima colonna (indice zero) contenga i voti, mentre la seconda (indice uno) contenga i nomi delle rispettive materie; ogni riga memorizzerà quindi nella colonna zero il voto e nella colonna uno la rispettiva materia.
All'interno del ListBox verranno visualizzati prima i nomi quindi i voti delle quattro materie digitate.

Operatori orientati ai bit

Si applicano a dati di tipo intero per ricavare o modificarne il valore logico dei singoli bit.

Operatore	Descrizione
&	AND bit a bit
|	OR bit a bit
~	NOT bit a bit
^	XOR bit a bit
>>	Shift destro bit a bit
<<	Shift sinistro bit a bit

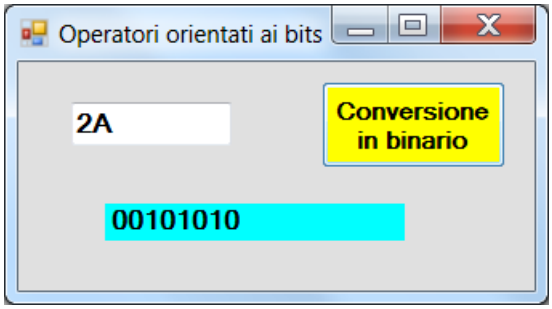
Operatore & per la verifica del valore di un bit di una variabile

Supponiamo che una variabile di tipo byte contenga il numero esadecimale 0x5B, il cui equivalente binario è 01011011, e che si voglia ricavare il valore logico del bit D0 (l'LSB).

Occorre allora realizzare l'AND bit a bit tra la variabile stessa ed una costante intera pari ad uno, avente quindi tutti i bit nulli, tranne il bit D0:

```
D7 D6 D5 D4 D3 D2 D1 D0
 0  1  0  1  1  0  1  1 → contenuto della variabile
 0  0  0  0  0  0  0  1 → costante
 --------------------------------
 0  0  0  0  0  0  0  1 → risultato della AND bit a bit
```

Si ricorda che nella funzione booleana prodotto logico (AND) comanda lo zero logico; è sufficiente che uno dei due operandi valga zero per avere come risultato zero (esattamente come nel prodotto aritmetico).
Si osserva quindi che il risultato della AND tra variabile e costante è pari a zero se il bit D0 della variabile vale zero, mentre è diverso da zero – come nel caso dell'esempio preedente – se il bit D0 vale uno.
Si procede in modo simile per verificare il valore logico di uno qualsiasi degli altri sette bit; ad esempio per verificare il bit D4 occore fare la AND della variabile con la costante esadecimale 0x10 (0001000_2) che ha sette bit nulli e il bit D4 ad uno logico:e il risultato della AND è nullo il bit D4 della variabile è a zero logico, altrimenti è a uno logico.
Si può anche ragionare in decimale: per verificare D0 la costante con cui realizzare la AND vale $2^0 = 1$, per verificare D4 la costante vale $2^4 = 16$, per testare D7 $2^7 = 128$ e così via.
Proviamo allora a convertire in binario un numero intero utilizzando l'operatore & (AND).
Il programma richiede la digitazione nel TextBox di un numero esadecimale tra 0 ed FF e visualizza nella Label l'equivalente binario a 8 bit.



Creare un nuovo progetto e salvarlo con nome **OperatoriBitWise**.
Aggiungere al Form un controllo Button, una Label e un TextBox.
Proprietà Name:
TextBox: txtEsadec
Label: lblBinario
Button: btnBinario

La classe **NumberStyles** necessaria per gestire i numeri esadecimali è contenuta nel namespace System.Globalization che dovrà essere aggiunto, mediante la direttiva **using**, agli altri namespace del progetto con la seguente istruzione:

```
using System.Globalization;
```

Vediamo il codice per la conversione in binario:

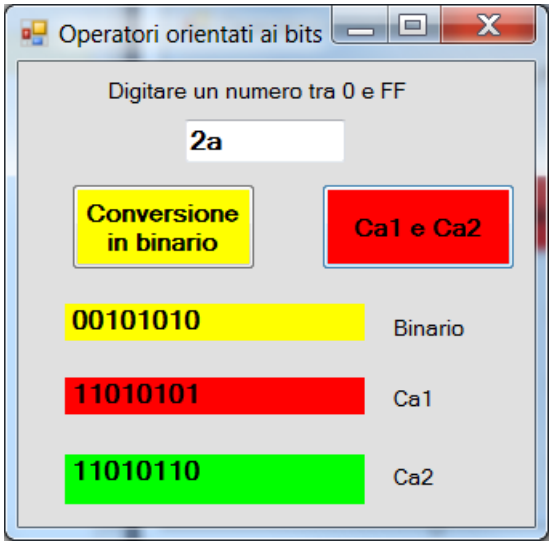
```
private void btnBinario_Click(object sender, EventArgs e)
    {
       string numeroString = txtEsadec.Text;
       string binario = "";
       // conversione da stringa esadecimale a int mediante il
       // metodo Parse della classe int
       int numeroInt = int.Parse(numeroString, NumberStyles.HexNumber);
       // conversione in binario con operatore & (AND)
       for (int i = 1; i <= 128; i = i*2)
       {
          if ((numeroInt & i) == 0)
             binario = '0' + binario;
          else
             binario = '1' + binario;
```

```
        }
        lblBinario.Text = binario;
    }
```

Per convertire in un numero intero la stringa rappresentante un numero esadecimale si usa il metodo ***Parse()*** della classe **int**, fornendo al metodo la variabile contenente la stringa da convertire e l'informazione sulla base del numero (**NumberStyles**.HexNumber).
Per convertire il numero in una striga binaria si usa il ciclo **for**. La variabile contatore ***i*** viene inizializzata ad uno e raddoppiata ad ogni loop generando gli otto valori 1, 2, 4, 8, 16, 32, 64, 128 necessari per realizzare le otto AND con la variabile *numeroInt*.
L'istruzione **if** ((numeroInt & i) == 0) effettua la AND tra la variabile *numeroInt* e la variabile ***i***: se il risultato è uguale a zero significa che il relativo bit della variabile *numeroInt* ha valore zero e quindi alla stringa *binario* viene concatenato il carattere **'0'**; se invece la condizione all'interno dell' **if** è falsa, significa che il relativo bit vale uno e quindi alla stringa *binario* viene concatenato il carattere **'1'**.

Operatore ~ per negare tutti i bit di una variabile

L'operatore ~ (NOT) nega tutti i bit di un variabile intera.
Come esempio, modifichiamo il progetto precedente in modo che del numero binario ottenuto vengano anche mostrati il complemento a 1 e a 2.



Aprire il progetto precedente e aggiungere al Form un secondo pulsante (btnCa1) ed altre due Label (lblCa1 e lblCa2).
Il codice all'interno del metodo btnCa1_Click() è simile a quello eseguito quando si clicca il pulsante per la conversione in binario; la differenza è che prima di effettuare la conversione in binario del numero memorizzato nella variabile *numero_int*, viene fatto il complemento a 1 della stessa variabile - mediante l'operatore ~ (NOT) - e successivamente il complemento a due, sommando uno alla variabile contenente il ca1.

```
private void btnCa1_Click(object sender, EventArgs e)
    {
        string numeroString = txtEsadec.Text;
        string binario = "";
        // conversione da stringa esadecimale a int mediante il
        // metodo Parse della classe int
        int numeroInt = int.Parse(numeroString, NumberStyles.HexNumber);
        // complemento a 1 tramite operatore ~ (NOT)
        numeroInt = ~ numeroInt;
        // conversione in binario con operatore & (AND)
        for (int i = 1; i <= 128; i = i * 2)
        {
            if ((numeroInt & i) == 0)
                binario = "0" + binario;
            else
                binario = "1" + binario;
        }
```

```
        lblCa1.Text = binario;
        // ricava ca2 sommando 1al ca1
        numeroInt ++;
        // conversione in binario con operatore & (AND)
        binario = "";
        for (int i = 1; i <= 128; i = i * 2)
        {
            if ((numeroInt & i) == 0)
                binario = "0" + binario;
            else
                binario = "1" + binario;
        }
        lblCa2.Text = binario;
    }
```

Operatori And (&) e Or (|) per modificare uno o più bit di una variabile

Supponiamo che una variabile di tipo byte contenga il numero esadecimale 0x5B, il cui equivalente binario è 01011011, e che si voglia ***azzerare*** il bit D0 (l'LSB) senza modificare il valore degli altri sette bit.
Occorre allora realizzare l'AND bit a bit tra la variabile stessa ed una costante intera pari a 0xFE, cioè una costante avente quindi tutti i bit a uno logico, tranne il bit D0:

D7	D6	D5	D4	D3	D2	D1	D0	
0	1	0	1	1	0	1	1	→ contenuto della variabile
1	1	1	1	1	1	1	0	→ costante (0xFE)
0	1	0	1	1	0	1	0	→ risultato della AND bit a bit

Si osserva quindi che il risultato della **AND** tra variabile e costante è un numero binario in cui i sette bit più significativi della variabile hanno mantenuto il valore iniziale, mentre il bit D0 è stato azzerato.
Si procede in modo simile per azzerare il valore logico di uno qualsiasi degli altri sette bit; ad esempio per azzerare il bit D5 occore fare la AND della variabile con la costante esadecimale 0xDF (11011111_2) che ha sette bit pari ad uno e il solo bit D5 ad uno logico.
Supponiamo ora che una variabile di tipo byte contenga il numero esadecimale 0xAC, il cui equivalente binario è 10101100, e che si voglia ***settare*** a uno logico il bit D0 (l'LSB) senza modificare il valore degli altri sette bit.
Occorre allora realizzare la **OR** (|) bit a bit tra la variabile stessa ed una costante intera pari ad uno, avente quindi tutti i bit nulli, tranne il bit D0:

D7	D6	D5	D4	D3	D2	D1	D0	
1	0	1	0	1	1	0	0	→ contenuto della variabile
0	0	0	0	0	0	0	1	→ costante
1	0	1	0	1	1	0	1	→ risultato della OR bit a bit

Si ricorda che nella funzione booleana somma logica (OR) comanda l'uno logico: è sufficiente che uno dei due operandi valga uno per avere come risultato uno.
Si osserva quindi che il risultato della OR tra variabile e costante è un numero binario in cui i sette bit più significativi della variabile hanno mantenuto il valore iniziale, mentre il bit D0 è stato forzato ad uno.
Si procede in modo simile per forzare ad uno il valore logico di uno qualsiasi degli altri sette bit; ad esempio per settare il bit D7 occore fare la OR della variabile con la costante esadecimale 0x80 (10000000_2) che ha sette bit pari a zero e il solo bit D7 ad uno logico.

Operatori >> e << per shiftare i bit di una variabile

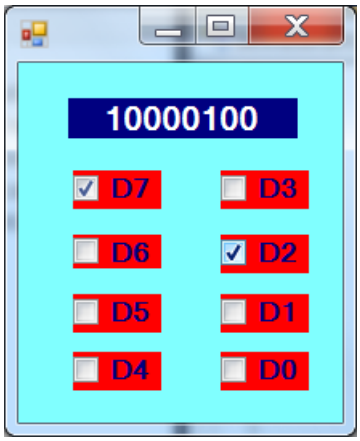
Supponiamo che una variabile di tipo byte contenga il numero esadecimale 0x5B, il cui equivalente binario è 01011011, e che si vogliano ***shiftare*** (far scorrere) a destra i bit della variabile di una posizione (si noti come il bit mancante - D7 - venga automaticamente rimpiazzato da uno zero):

D7	D6	D5	D4	D3	D2	D1	D0	
0	1	0	1	1	0	1	1	→ contenuto della variabile
0	1	1	1	1	1	1	1	→ variabile shiftata a destra di una posizione

Se invece si volesse shiftare tale variabile di due posizioni a sinistra:

D7	D6	D5	D4	D3	D2	D1	D0	
0	1	0	1	1	0	1	1	→ contenuto della variabile
0	1	1	0	1	1	0	0	→ variabile shiftata a sinistra di due posizioni

Anche in questo caso i due bit mancanti – D1 e D0 – vengono automaticamente rimpiazzati da zeri.
Come esempio di applicazione degli operatori AND, OR e shift vediamo un programma che permetta di



settare/resettare gli otto bit di una variabile di tipo byte, ognuno indipendentemente dagli altri.
Creare un nuovo progetto, salvarlo con nome **SetResetBit** e aggiungere al Form una Label (lblNumero) e otto controlli **CheckBox** (chkD7, chkD6, chkD5, chkD4, chkD3, chkD2, chkD1, chkD0).
Assegnare alla proprietà Text di lblNumero una stringa formata da otto zeri.
I controlli CheckBox sono simili ai RadioButton ma, a differenza di questi ultimi, possono essere selezionati e deselezionati indipendentemente l'uno dall'altro. Per esempio, con tutti i CheckBox selezionati la Label visualizzerà il numero binario 11111111, mentre nella figura a fianco, con i soli CheckBox D7 e D2 selezionati, viene selezionato il numero 10000100.
Sono quindi permesse tutte le 256 combinazioni(numeri) possibili con otto bit.

Analizziamo ora il codice presente all'interno del metodo chkD7_CheckedChanged(). L'evento CheckedChanged del CheckBox chkD7, analogamente a quanto visto per i controlli RadioButton, si verifica ogni volta che cambia lo stato di selezione del CheckBox.
L'istruzione **if** (chkD7.Checked) testa la proprietà Checked del CheckBox, se è true (CheckBox selezionato) l'istruzione:

```
numero = Convert.ToByte(numero | 0x80);
```

setta a uno logico il bit D7 della variabile *numero* facendo la OR di quest'ultima con il numero esadecimale ox80 (10000000_2).
Se la proprietà Checked del CheckBox è invece false (CheckBox deselezionato) l'istruzione:

```
numero = Convert.ToByte(numero & 0x7F);
```

azzera il bit D7 della variabile *numero* facendo la AND di quest'ultima con il numero esadecimale ox7F (01111111_2).
Successivamente, per visualizzare in binario il contenuto della variabile *numero* occorre convertire quest'ultima in una stringa binaria. Per operare la conversione utilizziamo, all'interno di un ciclo **for** eseguito otto volte, l'operatore AND unitamente all'operatore Shift a destra (>>).
Quando inizialmente la variabile contatore i assume il valore zero, allora l'istruzione:

```
numero >> i
```

shifterà di zero posizioni a destra i bit della variabile *numero*, il cui contenuto rimarrà quindi inalterato. Successivamente facendo la AND (&) con la costante uno otterremo un numero nullo se D7 vale zero - e quindi alla stringa vuota sarà concatenato il carattere '0' - altrimenti verrà concatenato il carattere '1'.
Al loop successivo, con i = 1, shiftando la variabile *numero* di una posizione a destra il bit D1 sostituirà D0 e quindi la AND con la costante uno fornirà il valore logico del bit D1, quindi con i = 2 sarà D2 a diventare il bit meno significativo e così via per i rimanenti bit fino a D7.
Gli altri sette metodi funzionano nello steso modo, permettendo di azzerare/settare gli altri sette bit del dato visualizzato nella Label; a cambiare saranno le costanti intere usate con gli operatori AND ed OR per azzerare/settare il bit interessato.

```
// variabili globali
        byte numero = 0;
        string numString = "";

        private void chkD7_CheckedChanged(object sender, EventArgs e)
        {
            if (chkD7.Checked)
                numero = Convert.ToByte(numero | 0x80); // forza D7 = 1
            else
                numero = Convert.ToByte(numero & 0x7F); // azzera D7
       // conversione in binario
            numString = "";
            for (int i = 0; i <= 7; i++)
                if (((numero >> i) & 1) == 0)
                    numString = '0' + numString;
                else
                    numString = '1' + numString;
         //visualizzazione numero binario
            lblNumero.Text = numString;
        }
        private void chkD0_CheckedChanged(object sender, EventArgs e)
        {
            if (chkD0.Checked)
                numero = Convert.ToByte(numero | 1);
            else
                numero = Convert.ToByte(numero & 0xFE);
            numString = "";
            for (int i = 0; i <= 7; i++)
                if (((numero >> i) & 1) == 0)
                    numString = '0' + numString;
                else
                    numString = '1' + numString;
            lblNumero.Text = numString;
        }

        private void chkD1_CheckedChanged(object sender, EventArgs e)
        {
            if (chkD1.Checked)
                numero = Convert.ToByte(numero | 2);
            else
                numero = Convert.ToByte(numero & 0xFD);
            numString = "";
            for (int i = 0; i <= 7; i++)
```

```
        if (((numero >> i) & 1) == 0)
            numString = '0' + numString;
        else
            numString = '1' + numString;
    lblNumero.Text = numString;
}

private void chkD2_CheckedChanged(object sender, EventArgs e)
{
    if (chkD2.Checked)
        numero = Convert.ToByte(numero | 4);
    else
        numero = Convert.ToByte(numero & 0xFB);
    numString = "";
    for (int i = 0; i <= 7; i++)
        if (((numero >> i) & 1) == 0)
            numString = '0' + numString;
        else
            numString = '1' + numString;
    lblNumero.Text = numString;
}

private void chkD3_CheckedChanged(object sender, EventArgs e)
{
    if (chkD3.Checked)
        numero = Convert.ToByte(numero | 8);
    else
        numero = Convert.ToByte(numero & 0xF7);
    numString = "";
    for (int i = 0; i <= 7; i++)
        if (((numero >> i) & 1) == 0)
            numString = '0' + numString;
        else
            numString = '1' + numString;
    lblNumero.Text = numString;
}

private void chkD4_CheckedChanged(object sender, EventArgs e)
{
    if (chkD4.Checked)
        numero = Convert.ToByte(numero | 0x10);
    else
        numero = Convert.ToByte(numero & 0xEF);
    numString = "";
    for (int i = 0; i <= 7; i++)
        if (((numero >> i) & 1) == 0)
            numString = '0' + numString;
        else
            numString = '1' + numString;
    lblNumero.Text = numString;
}
```

```
private void chkD5_CheckedChanged(object sender, EventArgs e)
 {
    if (chkD5.Checked)
       numero = Convert.ToByte(numero | 0x20);
    else
       numero = Convert.ToByte(numero & 0xDF);
    numString = "";
    for (int i = 0; i <= 7; i++)
       if (((numero >> i) & 1) == 0)
          numString = '0' + numString;
       else
          numString = '1' + numString;
    lblNumero.Text = numString;
 }

 private void chkD6_CheckedChanged(object sender, EventArgs e)
 {
    if (chkD6.Checked)
       numero = Convert.ToByte(numero | 0x40);
    else
       numero = Convert.ToByte(numero & 0xBF);
    numString = "";
    for (int i = 0; i <= 7; i++)
       if (((numero >> i) & 1) == 0)
          numString = '0' + numString;
       else
          numString = '1' + numString;
    lblNumero.Text = numString;
  }
```

Metodi C#

Un *metodo* è una porzione di codice che viene richiamata tramite il suo nome o, in altre parole, un blocco di codice autonomo che esegue una elaborazione.
Nei programmi svolti in precedenza, ad esempio, uno dei metodi più frequentemente utilizzati ad inizio corso è stato il metodo WriteLine() utilizzato (*chiamato*) in un'Applicazione Console per visualizzare sul monitor un messaggio.
Successivamente abbiamo utilizzato metodi gestori di eventi, ad esempio i metodi eseguiti quando, in fase di esecuzione, si clicca su un pulsante.
Il linguaggio C# incorpora migliaia di metodi, ma spesso si ha la necessità di doverne scrivere di nuovi.
I metodi permettono di suddividere il programma in parti più ridotte, rendendolo più comprensibile; inoltre consentono il riutilizzo di segmenti di codice in programmi diversi.
Una volta scritto e definito un metodo, il codice al suo interno è eseguito ogni volta che il flusso del programma incontra una sua invocazione.
Definire un metodo significa creare e dotare di una nuova funzione il programma che si sta scrivendo.
Forma generale della struttura di un metodo:

```
[modificatori] tipo-di-ritorno nome-metodo ([lista parametri])     // intestazione del metodo
{
        blocco di codice del metodo
}
```

Le parti racchiuse tra parentesi quadre sono opzionali.
L'intestazione del metodo comprende:

- **modificatori** (opzionali): sono rappresentati da particolari parole chiave il cui significato sarà descritto in seguito
- **tipo-di-ritorno**: indica il tipo di dato che un metodo può restituire al blocco di codice chiamante; tale tipo è uno dei tipi standard del linguaggio visti in precedenza. Se il metodo non deve restituire alcun valore il tipo di ritorno è **void**
- **nome-metodo**: rappresenta il nome univoco attribuito al metodo. Solitamente il nome di un metodo inizia con una lettera maiuscola.
- **lista parametri** (opzionale): tipo e nome di una o più variabili separate da una virgola.

Se si riprende in considerazione l'ultimo progetto realizzato si osserva che tutte le istruzioni del programma sono contenute all'interno del metodo **btnMedia_Click**, la cui intestazione è costituita da:

- modificatore: **private**
- tipo-di-ritorno**: void**
- nome-metodo: **btnMedia_Click**
- ha due parametri di ingresso

Come primo e semplice esempio riapriamo il primo progetto Windows Application, il progetto CiaoMondo. Modifichiamo quindi il codice nel modo seguente:

```
namespace CiaoMondo
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent()
       }

      private void btnMessaggio_Click(object sender, EventArgs e)
      {
         // chiamata del metodo messaggio
         messaggio();
      }

     // definizione del metodo messaggio
      private void messaggio()
      {
         lblMessaggio.Text = "Benvenuti in laboratorio";
      }
   }
}
```

Ora il codice è stato suddiviso in due metodi: il gestore di evento ***btnMessaggio_Click*** e il metodo ***messaggio*** che stampa "Benvenuti in laboratorio" all'interno della Label.
In primo luogo esaminiamo la definizione del metodo ***messaggio***.
La parola chiave **void** indica che il metodo ***messaggio*** non restituisce alcun valore e le parentesi () vuote successive al nome del metodo indicano che il metodo non ha parametri d'ingresso.
Il corpo del metodo ***messaggio*** è costituito da un'unica istruzione che stampa "Benvenuti in laboratorio" all'interno della Label.
Il codice all'interno del metodo ***messaggio*** non deve necessariamente terminare con l'istruzione **return** essendo il metodo di tipo **void**, tuttavia è possibile chiudere il codice scrivendo tale istruzione:

```
private void messaggio()
  {
     lblMessaggio.Text = "Benvenuti in laboratorio";
     return;
  }
```

Occorre chiarire che le istruzioni all'interno del metodo ***messaggio*** non vengono eseguite spontaneamente, ma solo dopo aver eseguito la chiamata all'interno del metodo btnMessaggio_Click.
Al termine dell'esecuzione del metodo ***messaggio***, l'esecuzione del programma torna al metodo gestore di evento btnMessaggio_Click, più precisamente all'istruzione successiva alla chiamata del metodo ***messaggio***: non essendoci nessuna ulteriore istruzione dopo quella di chiamata, il metodo btnMessaggio_Click termina la propria esecuzione.

Metodi con parametri - Metodi che restituiscono un valore

Come detto in precedenza, le parentesi tonde successive al nome del metodo nell'intestazione contengono i ***parametri del metodo***.
I parametri sono informazioni fornite a un metodo, in modo che possa effettuare le proprie operazioni.
A differenza del metodo ***messaggio***, che per svolgere il proprio compito non necessita di ulteriori informazioni, spesso si definiscono metodi che necessitano di informazioni in ingresso (di parametri) per poter effettuare le proprie operazioni.
Inoltre, mentre il metodo ***messaggio*** è stato dichiarato **void** in quanto non restituisce alcun valore al codice chiamante, esistono metodi che devono restituire un valore (un numero intero, un numero reale, una stringa, etc.).
Si pensi ad esempio a un metodo che restituisce il quadrato di un numero: tale metodo deve conoscere il numero da elevare al quadrato (richiede quindi un parametro) e deve restituire in uscita un valore (il quadrato del numero).
Analogamente un metodo che calcola il parallelo tra due resistenze deve conoscere i valori delle due resistenze (richiede quindi due parametri) e deve restituire in uscita il valore del parallelo.

Come esempio di metodo richiedente un parametro e che restituisce un valore, riapriamo il progetto ***SetResetBit*** e modifichiamolo in modo che la conversione in binario di un numero intero sia realizzata da un metodo che abbia come parametro d'ingresso il numero da convertire e che restituisca una stringa che ne contiene la rappresentazione binaria .
Modifichiamo il codice nel modo seguente:

```
// variabili globali
   byte numero = 0;


   private void chkD7_CheckedChanged(object sender, EventArgs e)
   {
      if (chkD7.Checked)
         numero = Convert.ToByte(numero | 0x80);
      else
         numero = Convert.ToByte(numero & 0x7F);
      // chiama metodo binario
      lblNumero.Text = binario(numero);
   }

   private void chkD0_CheckedChanged(object sender, EventArgs e)
   {
      if (chkD0.Checked)
```

```
            numero = Convert.ToByte(numero | 1);
        else
            numero = Convert.ToByte(numero & 0xFE);
        // chiama metodo binario
        lblNumero.Text = binario(numero);
    }

    private void chkD1_CheckedChanged(object sender, EventArgs e)
    {
        if (chkD1.Checked)
            numero = Convert.ToByte(numero | 2);
        else
            numero = Convert.ToByte(numero & 0xFD);
        // chiama metodo binario
        lblNumero.Text = binario(numero);
    }

    private void chkD2_CheckedChanged(object sender, EventArgs e)
    {
        if (chkD2.Checked)
            numero = Convert.ToByte(numero | 4);
        else
            numero = Convert.ToByte(numero & 0xFB);
        // chiama metodo binario
        lblNumero.Text = binario(numero);
    }

    private void chkD3_CheckedChanged(object sender, EventArgs e)
    {
        if (chkD3.Checked)
            numero = Convert.ToByte(numero | 8);
        else
            numero = Convert.ToByte(numero & 0xF7);
        // chiama metodo binario
        lblNumero.Text = binario(numero);
    }

    private void chkD4_CheckedChanged(object sender, EventArgs e)
    {
        if (chkD4.Checked)
            numero = Convert.ToByte(numero | 0x10);
        else
            numero = Convert.ToByte(numero & 0xEF);
        // chiama metodo binario
        lblNumero.Text = binario(numero);
    }

    private void chkD5_CheckedChanged(object sender, EventArgs e)
    {
        if (chkD5.Checked)
            numero = Convert.ToByte(numero | 0x20);
        else
            numero = Convert.ToByte(numero & 0xDF);
```

```
        // chiama metodo binario
        lblNumero.Text = binario(numero);
    }

    private void chkD6_CheckedChanged(object sender, EventArgs e)
    {
        if (chkD6.Checked)
            numero = Convert.ToByte(numero | 0x40);
        else
            numero = Convert.ToByte(numero & 0xBF);
        // chiama metodo binario
        lblNumero.Text = binario(numero);
    }

// metodo binario per conversione intero tra 0 e 255 in binario
    private string binario( byte num )
    {
        string numString = "";
        for (int i = 0; i <= 7; i++)
            if (((num >> i) & 1) == 0)
                numString = '0' + numString;
            else
                numString = '1' + numString;
        /* fa si che il metodo binario restituisca in uscita il
         * contenuto della variabile numString */
        return numString;
    }
```

Cominciamo con l'esaminare l'intestazione del metodo ***binario***: il metodo è stato dichiarato di tipo **string** in quanto restituisce una stringa, quella contenuta nella variabile *numString* che segue la parola chiave **return**.
E' opportuno chiarire che un metodo può restituire solamente un unico valore.

L'intestazione del metodo ***binario***, inoltre, contiene un nome di parametro (***num***) e un tipo di dati (**byte**), in modo che il valore passato da una qualsiasi delle otto chiamate del metodo ***binario*** nei gestori degli eventi CheckedChanged possa essere memorizzato in una variabile utilizzabile all'interno del metodo ***binario***.
Analizziamo ad esempio la chiamata del metodo ***binario*** presente all'interno del metodo gestore di evento chkD6_CheckedChanged: il contenuto della variabile **byte** di nome ***numero*** viene passata come argomento al metodo ***binario***. Il valore di tale proprietà viene quindi passato alla variabile **byte** di nome ***num***, il nome di parametro nell'intestazione del metodo ***binario***.
In pratica, quando viene eseguita l'istruzione di chiamata del metodo ***binario***, il contenuto della variabile ***numero*** viene copiato nella variabile ***num***.
La stringa restituita dal metodo ***binario***, che rappresenta l'equivalente binario del numero intero passato al metodo, sarà assegnata alla proprietà Text – anch'essa di tipo **string** – della Label di nome lblNumero, permettendo così la visualizzazione del numero binario ottenuto.
Analogamente per gli altri sette metodi gestori degli eventi CheckedChanged.
Si può quindi verificare uno dei vantaggi nell'utilizzo dei metodi: invece di dover riscrivere otto volte il codice per la conversione in binario, è sufficiente scriverlo un'unica volta nel metodo ***binario*** e poi chiamare quest'ultimo otto volte.

Metodi con più parametri

Vediamo ora un'applicazione che utilizza tre metodi ognuno dei quali possiede tre parametri.
Riapriamo il progetto SetResetBit e modifichiamolo introducendo i tre metodi ***read***, ***set*** e ***reset***.
Il metodo ***read*** utilizza due parametri: il primo (di nome *dato*) riceve come argomento un intero di tipo **byte**, il secondo (di nome *numbit*) riceve come argomento un intero nell'intervallo 0-7 che identifica uno degli otto bit del dato passato; bit il cui valore logico (zero o uno) sarà restituito dal metodo stesso.
Il metodo ***set*** utilizza due parametri: il primo (di nome *dato*) riceve come argomento un intero di tipo **byte**, il secondo (di nome *numbit*) riceve come argomento un intero nell'intervallo 0-7 che identifica uno degli otto bit del dato passato; tale bit sarà settato a uno logico senza alterare i valori logici dei rimanenti sette bit.
Il metodo ***reset*** utilizza due parametri: il primo (di nome *dato*) riceve come argomento un intero di tipo **byte**, il secondo (di nome *numbit*) riceve come argomento un intero nell'intervallo 0-7 che identifica uno degli otto bit del dato passato; tale bit sarà azzerato senza alterare i valori logici dei rimanenti sette bit.

Il metodo ***read*** viene chiamato otto volte all'interno del ciclo *for* presente nel metodo ***binario***, realizzando la conversione in binario del numero intero ad otto bit passato come parametro al metodo stesso.
I metodi ***set*** e ***reset*** sono usati negli otto metodi gestori degli eventi CheckedChanged abbinati agli otto controlli CheckBox: quando uno degli otto controlli CheckBox viene selezionato viene chiamato il metodo ***set*** passandogli come primo argomento il contenuto della variabile globale *numero* e come secondo argomento, separato da una virgola, il numero del bit della variabile in questione da settare a uno logico.
Analogamente, quando un controllo CheckBox viene deselezionato viene chiamato il metodo ***reset*** passandogli come primo argomento il contenuto della variabile globale *numero* e come secondo argomento, separato da una virgola, il numero del bit della variabile in questione da forzare a zero logico.
Come detto in precedenza, quando si dichiara un metodo con più parametri questi sono separati da virgole; analogamente nella chiamata del metodo gli argomenti sono separati da virgole e il loro ordine nella chiamata deve corrispondere all'ordine dei parametri nell'intestazione del metodo.
Si vedano ad esempio la dichiarazione del metodo ***read*** e la sua chiamata:

```
byte read(byte dato, byte numbit)  // dichiarazione del metodo read
read(num,i)  // chiamata del metodo read
```

La prima variabile nella chiamata del metodo è *num*, quindi il suo valore viene copiato nel primo parametro dell'intestazione (la variabile *dato*). Analogamente il valore del secondo argomento nella chiamata – la variabile *i* – viene copiato nel secondo parametro dell'intestazione (la variabile *numbit*).
Il codice risultante sarà quindi il seguente:

```
// variabile globale
    byte numero = 0;

    byte read(byte dato, byte numbit)
    {
       /* metodo che restituisce il valore logico (0 oppure 1) di uno degli otto bit
        * (numerati da 0 a 7) di una variabile byte  */
       return (byte)((dato & (1 << numbit)) >> numbit);
    }

    byte set(byte dato, byte numbit)
    {
       /* metodo che setta a 1 logico uno dei bit
        * (numerati da 0 a 15) di una variabile intera */
       return (byte)(dato | (1 << numbit));
    }
```

```
byte reset(byte dato, byte numbit)
 {
    /* metodo che forza a 0 logico uno dei bit
     * (numerati da 0 a 15) di una variabile intera */
    return (byte)(dato & ~(1 << numbit));
 }

 /* metodo binario per conversione intero tra 0 e 255 in binario
  * utilizzante il metodo read   */
 private string binario(byte num)
 {
    string numString = "";
    for (byte i = 0; i <= 7; i++)
        numString = read(num,i) + numString;
    /* fa si che il metodo binario restituisca in uscita il
     * contenuto della variabile numString */
    return numString;
 }

 private void chkD7_CheckedChanged(object sender, EventArgs e)
 {
    if (chkD7.Checked)
       numero = set(numero,7);
    else
       numero = reset(numero, 7);
    // chiama metodo binario
    lblNumero.Text = binario(numero);
 }

 private void chkD0_CheckedChanged(object sender, EventArgs e)
 {
    if (chkD0.Checked)
       numero = set(numero, 0);
    else
       numero = reset(numero,0);
    // chiama metodo binario
    lblNumero.Text = binario(numero);
 }

 private void chkD1_CheckedChanged(object sender, EventArgs e)
 {
    if (chkD1.Checked)
       numero = set(numero, 1);
    else
       numero = reset(numero,1);
    // chiama metodo binario
    lblNumero.Text = binario(numero);
 }

 private void chkD2_CheckedChanged(object sender, EventArgs e)
 {
```

```
    if (chkD2.Checked)
        numero = set(numero, 2);
    else
        numero = reset(numero,2);
    // chiama metodo binario
    lblNumero.Text = binario(numero);
}

private void chkD3_CheckedChanged(object sender, EventArgs e)
{
    if (chkD3.Checked)
        numero = set(numero,3);
    else
        numero = reset(numero,3);
    // chiama metodo binario
    lblNumero.Text = binario(numero);
}

private void chkD4_CheckedChanged(object sender, EventArgs e)
{
    if (chkD4.Checked)
        numero = set(numero,4);
    else
        numero = reset(numero,4);
    // chiama metodo binario
    lblNumero.Text = binario(numero);
}

private void chkD5_CheckedChanged(object sender, EventArgs e)
{
    if (chkD5.Checked)
        numero = set(numero,5);
    else
        numero = reset(numero,5);
    // chiama metodo binario
    lblNumero.Text = binario(numero);
}

private void chkD6_CheckedChanged(object sender, EventArgs e)
{
    if (chkD6.Checked)
        numero = set(numero,6);
    else
        numero = reset(numero,6);
    // chiama metodo binario
    lblNumero.Text = binario(numero);
}
```

Introduzione alle classi e agli oggetti

Una **classe** è una categoria o un gruppo di **oggetti** (intendendo con oggetti anche gli esseri viventi) che hanno attributi simili e comportamenti analoghi.
Per esempio, ciascun lettore di questi appunti è una persona e, come tale, condivide alcune caratteristiche comuni a tutte le persone, quali un nome, un'altezza, un peso, un genere, un'età e così via.
I linguaggi di programmazione, compreso Visual C#, utilizzano classi per rappresentare una persona, un luogo, un oggetto o un concetto.
Di conseguenza, nel gergo della programmazione, ciascuna persona è un oggetto della *classe **Persona***. Una classe è quindi uno schema, o un modello, per un oggetto e un *oggetto* è un'**istanza** di una classe.
Per esempio, se un'aula scolastica contiene 29 studenti e un insegnante, sono presenti 30 oggetti della *classe **Persona*** o, in altre parole, 30 istanze della classe ***Persona***.
Di nuovo, nome, altezza, peso, genere ed età di ciascuno possono essere diversi, ma essendo ogni persona presente nell'aula un oggetto della medesima classe ***Persona***, essa condivide con le altre alcune caratteristiche comuni quali nome, altezza, peso, genere ed età.
Una classe può contenere al suo interno, oltre che dati, anche parte di codice (metodi): questo è il meccanismo *dell'**incapsulamento*** che riunisce insieme codice e dati da esso manipolati e che mette entrambi al sicuro da interferenze o errati utilizzi.
In termini ingegneristici il termine "***scatola nera***" indica qualsiasi dispositivo di cui non si conoscano i meccanismi interni di funzionamento, ma le cui interazioni con il mondo esterno siano ben definite.
Ad esempio, per molti guidatori un'automobile è una "scatola nera" nel senso che essi non sanno nulla del suo funzionamento interno; sottoponendo però pedali, pulsanti, etc. agli opportuni stimoli in ingresso, l'automobile svolge il suo compito trasportando l'automobilista da una parte all'altra.
Gli sviluppatori di software, man mano che i programmi divennero sempre più complessi, impararono a gestirne la complessità incapsulando le elaborazioni più frequenti, creando quindi "scatole nere" software da poter utilizzare senza occuparsi di ciò che avviene all'interno.
Le scatole nere con cui sono costruiti i programmi vengono chiamate **oggetti**.
Quindi in un linguaggio di programmazione a oggetti (**O**bject **O**riented **P**rogramming) il codice e i dati possono essere raggruppati in modo da creare un oggetto, cioè una sorta di scatola nera.
La struttura interna di un oggetto risulta quindi nascosta al programmatore che lo utilizza; tale programmatore deve solamente apprendere il comportamento dell'oggetto.
In alcuni casi, però, è necessario progettare oggetti, oltre che utilizzarli.
Un oggetto è, in tutto e per tutto, una variabile di un tipo definito dall'utente, quindi ogni volta che si definisce un nuovo tipo di oggetto, si crea implicitamente un nuovo tipo di dato.

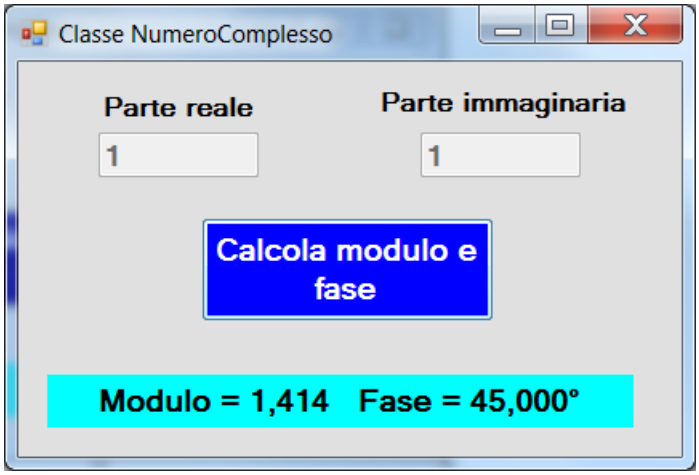
- In C# per creara un oggetto si deve innanzitutto definire la sua forma generale utilizzando la parola chiave ***class***.
- In C# ***class*** crea un nuovo tipo di dati che può essere usato per creare oggetti appartenenti alla classe
- Tutti gli elementi che compongono la definizione di una classe prendono il nome di **membri**
- All'interno dei membri si distinguono **dati** e **metodi**, dove un metodo è un'azione che gli oggetti di una certa classe possono compiere, mentre un dato rappresenta una *proprietà* di una classe; esso descrive un insieme di valori che la proprietà può avere
- E' anche possibile specificare il valore di default che un dato può avere
- Il nome della classe, per convenzione, è una parola con l'iniziale maiuscola
- Se tale nome è una parola composta a sua volta da più parole, allora si usa la notazione in cui tutte le iniziali di ogni parola sono scritte in maiuscolo (Es. GestioneClienti)
- Un dato il cui nome è costituito da una sola parola viene scritto sempre in caratteri minuscoli; se invece il nome consiste di più parole, allora esso viene scritto unendo tutte le parole che lo costituiscono, con la particolarità che la prima parola viene scritta in minuscolo, mentre le successive hanno la loro prima lettera in maiuscolo (Es. informazioneCliente)
- I nomi dei metodi sono scritti con gli stessi criteri adottati per i dati

Graficamente una classe viene rappresentata da un rettangolo. Il nome della classe appare vicino alla sommità del rettangolo e dopo di esso, separati tramite una linea orizzontale, vengono descritti i dati a essa appartenenti. La lista dei metodi è rappresentata graficamente sotto la lista dei dati e separata da questa tramite una linea orizzontale.
Come primo esempio, diamo la rappresentazione grafica della classe **NumeroComplesso**.

NumeroComplesso
double real **double imm**
double modulo() **double fase()**

Abbiamo quindi definito la classe **NumeroComplesso** che contiene al suo interno i ***dati*** relativi alla parte reale ***real*** e alla parte immaginaria ***imm*** e i due ***metodi*** utilizzati per il calcolo del modulo e della fase di un numero complesso.

- Una classe può contenere parti private e parti pubbliche. In generale, tutti i membri definiti all'interno di una classe sono privati.
- I dati e i metodi ***private*** non sono visibili da alcun altro metodo che non sia definito all'interno della classe. In assenza di indicazione (per default) si assume che il livello di protezione sia *private*
- Solo i metodi di una classe hanno accesso ai dati privati della classe in cui sono dichiarati
- Per rendere pubblica (ovvero accessibile da altre parti del programma) una parte della classe, è necessario dichiararla tale esplicitamente usando la parola chiave ***public***. Tutte le variabili e le funzioni definite dopo *public* possono essere utilizzate da tutte le altre funzioni del programma. Essenzialmente, la parte rimanente del programma accede a un oggetto utilizzando le sue funzioni pubbliche.
- Anche se è possibile avere variabili pubbliche, si deve in generale cercare di limitarne l'uso rendendo tutti i dati privati e controllandone l'accesso mediante funzioni pubbliche. E' comunque possibile definire funzioni *private* che possono essere richiamate solo dai membri della classe.



Creiamo quindi un nuovo progetto, con nome **classeNumeroComplesso**, che definisca e utilizzi la classe **NumeroComplesso** illustrata in precedenza.

Aggiungere al Form un controllo Button, quattro Label e due TextBox.
Proprietà Name:
Label: lblRisultato
Button: btnCalcola
TextBox: txtReale, txtImmaginaria

Nella **Finestra delle Proprietà** assegnare "1" alle proprietà Text dei due TextBox e poi disabilitare entrambi i TextBox selezionando il valore booleano **False** per la proprietà **Enabled** di entrambi i controlli: questo impedirà all'utente, in fase di esecuzione del programma, di modificare il contenuto dei due TextBox.
Fare infine doppio click sul pulsante ***btnCalcola*** per creare il metodo gestore d'evento btnCalcola_Click().
Cominciamo con creare la classe **NumeroComplesso.**

All'interno del namespace classeNumeroComplesso (i namespace sono contenitori di classi), scriviamo class NumeroComplesso seguita da una coppia di parentesi graffe al cui interno scriveremo i due dati e i due metodi che compaiono nella rappresentazione grafica della classe **NumeroComplesso** vista in precedenza.

```
namespace classeNumeroComplesso
{

// Classe Form1
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
      }

      private void btnCalcola_Click(object sender, EventArgs e)
      {
         /* dichiara oggetto della classe NumeroComplesso
          e richiama il metodo costruttore */
         NumeroComplesso  z = new NumeroComplesso();
         // chiama metodi pubblici della classe NumeroComplesso per calcolare modulo e fase in gradi
         lblRisultato.Text = string.Format("Modulo = {0:f3}   Fase = {1:f3}°", z.modulo(), z.fase() * 180 /
                                 Math.PI);
      }
   }

   // Classe NumeroComplesso
   class NumeroComplesso
   {
      // membri dati della classe
      double real = 1, imm = 1;

     // membri metodi della classe
      public double modulo()
      {
         // calcolo del modulo del numero complesso
         double mod = Math.Pow(this.real, 2) + Math.Pow(this.imm, 2);
         return Math.Sqrt(mod);
      }

      public double fase()
      {
         // calcolo della fase del numero complesso in radianti
         return Math.Atan2(this.imm, this.real);
      }
   }
}
```

Analizziamo il codice all'interno della classe **NumeroComplesso**.
Le prime istruzioni dichiarano i due membri dato privati *real* e *imm* che vengono inizializzati con due valori costanti.
Tali dati, essendo privati, potranno essere utilizzati solo dai metodi definiti all'interno della classe.

Le istruzioni successive definiscono due metodi pubblici (*modulo* e *fase*), utilizzabili anche all'interno della classe **Form1**, che non richiedono parametri in ingresso e che restituiscono, rispettivamente, il modulo e la fase del numero complesso avente come parte reale e parte immaginaria i valori assegnati, rispettivamente, ai membri dato *real* e *imm*.
Per il calcolo della fase vengono utilizzati i metodi Math.Sqrt e Math.Pow già visti nel progetto **EqSecondoGrado**, mentre per il calcolo della fase (in radianti) del numero complesso si ricorre al metodo Math.Atan2 che richiede due parametri in ingresso: il primo riceverà come argomento la parte immaginaria, il secondo la parte reale. Si osservi, all'interno dei due metodi della classe **NumeroComplesso**, l'utilizzo della parola chiave **this** che sta ad indicare l'oggetto corrente della classe.
Analizziamo ora il codice all'interno del metodo gestore di evento *btnCalcola_Click* della classe **Form1**.
La creazione di un nuovo oggetto in un programma C# avviene tramite l'operatore **new** che carica in memoria la struttura della classe.
Per esempio, quando si scrive:

```
NumeroComplesso  z = new NumeroComplesso();
```

si crea un nuovo oggetto **NumeroComplesso** di nome **z**. Al momento della creazione dell'oggetto il controllo viene passato a un particolare metodo chiamato ***costruttore***; metodo avente esattamente lo stesso nome della classe.
In C# non è necessario creare un metodo costruttore per la classe, infatti all'interno della classe **NumeroComplesso** non ne abbiamo scritto nessuno.
In generale, se non si scrive un proprio metodo costruttore, il compilatore ne creerà uno di default autonomamente. Sarà un costruttore molto semplice che si occuperà di inizializzare – nel caso in cui non sia già stato fatto – tutti i membri dato della classe.
Vedremo in seguito che esistono casi in cui sarà necessario scrivere dei propri costruttori.
I metodi costruttori seguono le stesse regole degli altri metodi C#, in particolare si possono scrivere costruttori che non richiedono parametri d'ingresso (come quello fornito di default dal compilatore e utilizzato in questo esempio) oppure metodi con parametri d'ingresso; in questo caso il compilatore non ne creerà alcuno di default.
Tornando ad esaminare il codice all'interno del metodo *btnCalcola_Click*, successivamente alla definizione dell'oggetto **z** della classe **NumeroComplesso** vengono chiamati i due metodi pubblici della classe digitando il nome dell'oggetto (**z**) seguito dal punto e selezionando quindi il nome del metodo desiderato (**modulo()** o **fase()**).
Quando, ad esempio, viene eseguita l'istruzione seguente:

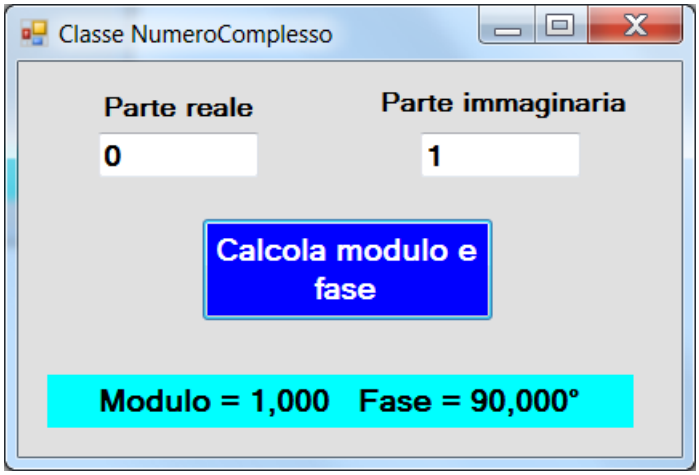
```
z.modulo()
```

all'interno del metodo modulo presente nella classe **NumeroComplesso** sarà calcolato il quadrato della parte reale dell'oggetto corrente (**z**) tramite l'istruzione Math.Pow(this.real, 2) e poi sommato con il quadrato della parte immaginaria dell'oggetto corrente (**z**) tramite l'istruzione Math.Pow(this.imm, 2).
In maniera simile opera l'istruzione **z.fase()** che restituisce la fase del numero complesso – dell'oggetto – **z** in radianti.
Nel nostro caso, essendo i due membri dato *real* e *imm* inizializzati al valore unitario, qualsiasi oggetto della classe, cioè qualsiasi numero complesso, avrà parte reale e parte immaginaria paria uno e quindi lo stesso modulo e la stessa fase.
Essendo la fase restituita dall'istruzione **z.fase()** espressa in radianti, all'interno del metodo **Format** della classe **string** viene operata la trasformazione da radianti a gradi.

Proviamo a modificare il progetto precedente in modo che sia possibile dichiarare oggetti – numeri complessi – con parte reale e parte immaginaria stabilite dall'utente del programma mediante digitazione nei due TextBox.
Cominciamo quindi con l'abilitare i due TextBox selezionando il valore booleano **False** per la proprietà **Enabled** di entrambi i controlli: questo permetterà all'utente, in fase di esecuzione del programma, di modificare il contenuto dei due TextBox inserendo parte reale e immaginaria desiderate.

Nella classe **NumeroComplesso** eliminiamo l'inizializzazione ad uno delle due variabili *real* e *imm* e scriviamo un nostro metodo costruttore con due parametri d'ingresso che ricevano come argomenti parte reale e parte immaginaria acquisite dai due TextBox.
Si osservi che:

- il metodo costruttore ha lo stesso nome della classe, è pubblico e non è di nessun tipo (neppure void)

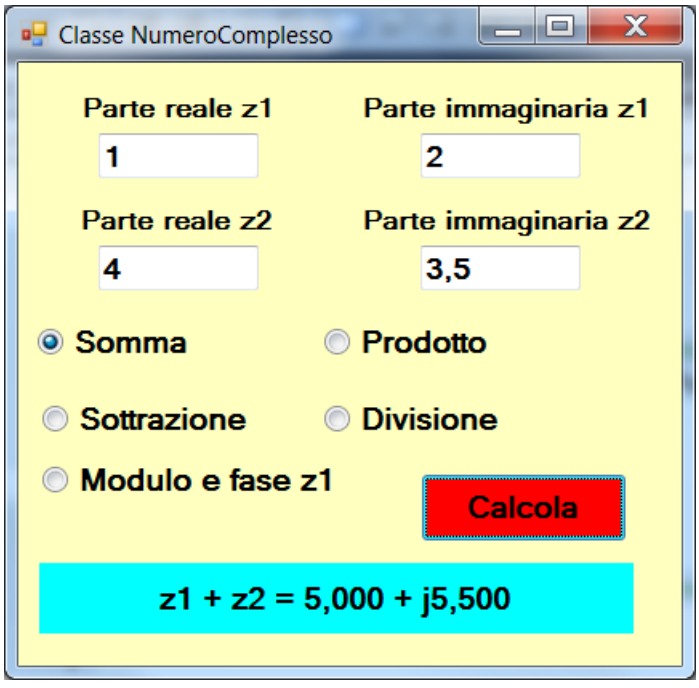


Il compito del costruttore è di inizializzare i due membri dato privati *real* e *imm* copiando in questi ultimi il contenuto dei suoi due parametri *cReal* e *cImm*.
Passando ad esaminare il metodo *btnCalcola_Click* vediamo che, nell'istruzione di dichiarazione dell'oggetto **z**, la chiamata del metodo costruttore avviene passando i due valori acquisiti dai TextBox (e salvati nelle variabili *a* e *b*) come argomenti ai due parametri d'ingresso del costruttore *cReal* e *cImm*.
Il costruttore provvede poi ad inizializzare i due membri dato *real* e *imm* dell'oggetto corrente (**z** nel nostro esempio) con i valori copiati nei suoi parametri.

```
namespace classeNumeroComplesso
{
   // Classe Form1
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
      }

      private void btnCalcola_Click(object sender, EventArgs e)
      {
         // acqusizione dati
         double a = Convert.ToDouble(txtReale.Text);
         double b = Convert.ToDouble(txtImmaginaria.Text);
         /* dichiara oggetto della classe NumeroComplesso
          e richiama il metodo costruttore con parametri */
         NumeroComplesso z = new NumeroComplesso(a, b);
         // chiama metodi pubblici della classe NumeroComplesso
         lblRisultato.Text = string.Format("Modulo = {0:f3}   Fase = {1:f3}°", z.modulo(), z.fase() * 180 /
                                 Math.PI);
      }
   }

// Classe NumeroComplesso
   class NumeroComplesso
   {
      // dati della classe
      double real, imm;

      // metodi della classe
      public double modulo()
      {
```

```
        // calcolo del modulo del numero complesso
        double mod = Math.Pow(this.real, 2) + Math.Pow(this.imm, 2);
        return Math.Sqrt(mod);
    }

    public double fase()
    {
        // calcolo della fase del numero complesso in radianti
        return Math.Atan2(this.imm, this.real);
    }

    // metodo costruttore
    public NumeroComplesso(double cReal, double cImm)
    {
        this.real = cReal;
        this.imm = cImm;
    }
  }
}
```

Proviamo infine ad aggiungere alla classe **NumeroComplesso** quattro metodi che realizzino le quattro operazioni aritmetiche sui numeri complessi. I quattro metodi - di nome *somma*, *sottrazione*, *prodotto* e *divisione* - richiedono come parametro d'ingresso e restituiscono in uscita un oggetto di tipo **NumeroComplesso** che rappresenta il risultato dell'operazione aritmetica in questione tra l'oggetto corrente (**this**) e l'oggetto passato al parametro d'ingresso.



Aggiungiamo quindi al Form altri due TextBox e cinque RadioButton che permettano di selezionare l'operazione da svolgere.

Proprietà Name:
Label: lblRisultato
Button: btnCalcola
TextBox: txtReal1, txtImm1, txtReal2, txtImm2
RadioButton: radSomma, radSottrazione, radProdotto, radDivisione, radModFase

Esaminiamo il codice all'interno del metodo *btnCalcola_Click*.
Se, ad esempio, si è selezionato il RadioButton per comandare la somma dei due numeri complessi, allora, dopo l'acquisizione delle parti reali e immaginarie dei due numeri complessi, vengono dichiarati (istanziati) due oggetti della classe **NumeroComplesso** di nome **z1** e **z2**.
Al costruttore della classe in questione vengono passati come argomenti i valori della parte reale e di quella immaginaria digitati per il numero z1 e per il numero z2.
L'istruzione:

```
z2 = z1.somma(z2);
```

chiama il metodo ***somma*** dell'oggetto **z1** passandogli come argomento l'oggetto **z2**;il metodo in questione restituisce in uscita un oggetto della classe **NumeroComplesso** che viene assegnato all'oggetto **z2**.

L'oggetto **z2**, quindi, dopo l'esecuzione dell'istruzione precedente conterrà il numero complesso risultante dalla somma dei due numeri(dei due oggetti) z1 e z2 digitati nei TextBox.
Vediamo ora il codice all'interno del metodo *somma* della classe **NumeroComplesso**.
Come già detto, la parola chiave **this** fa riferimento all'oggetto corrente, in questo caso l'oggetto **z1**, mentre il parametro **z** riceve come argomento l'oggetto **z2**; il resto del codice calcola le somme degli attributi ***real*** e ***imm*** dei due oggetti assegnandole agli omonimi membri dati dell'oggetto corrente e restituisce in uscita tale oggetto.
Gli altri metodi della classe **NumeroComplesso** funzionano in modo simile restituendo un oggetto che rappresenta il risultato delle altre operazioni aritmetiche tra i due numeri complessi.

```
namespace classeNumeroComplesso
{
   // Classe Form1
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
      }

      private void btnCalcola_Click(object sender, EventArgs e)
      {
         double modulo, fase, a, b, c, d;
         if (radSomma.Checked)
         {
            // acqusizione dati
            a = Convert.ToDouble(txtReal1.Text);
            b = Convert.ToDouble(txtImm1.Text);
            c = Convert.ToDouble(txtReal2.Text);
            d = Convert.ToDouble(txtImm2.Text);
            /* dichiara oggetto della classe NumeroComplesso
             e richiama il metodo costruttore con parametri */
            NumeroComplesso z1 = new NumeroComplesso(a, b);
            NumeroComplesso z2 = new NumeroComplesso(c, d);
            // chiama metodo pubblico della classe NumeroComplesso
            z2 = z1.somma(z2);
            modulo = z2.modulo();
            fase = z2.fase();
            lblRisultato.Text = string.Format("z1 + z2 = {0:f3} + j{1:f3}", modulo * Math.Cos(fase), modulo
                                 * Math.Sin(fase));
         }
         else if (radSottrazione.Checked)
         {
            // acquisizione dati
            a = Convert.ToDouble(txtReal1.Text);
            b = Convert.ToDouble(txtImm1.Text);
            c = Convert.ToDouble(txtReal2.Text);
            d = Convert.ToDouble(txtImm2.Text);
            /* dichiara oggetto della classe NumeroComplesso
             e richiama il metodo costruttore con parametri */
            NumeroComplesso z1 = new NumeroComplesso(a, b);
            NumeroComplesso z2 = new NumeroComplesso(c, d);
            // chiama metodo pubblico della classe NumeroComplesso
```

```
    z2 = z1.sottrazione(z2);
    modulo = z2.modulo();
    fase = z2.fase();
    lblRisultato.Text = string.Format("z1 - z2 = {0:f3} + j{1:f3}", modulo * Math.Cos(fase), modulo
                        * Math.Sin(fase));
}
else if (radProdotto.Checked)
{
    // acquisizione dati
    a = Convert.ToDouble(txtReal1.Text);
    b = Convert.ToDouble(txtImm1.Text);
    c = Convert.ToDouble(txtReal2.Text);
    d = Convert.ToDouble(txtImm2.Text);
    /* dichiara oggetto della classe NumeroComplesso
     e richiama il metodo costruttore con parametri */
    NumeroComplesso z1 = new NumeroComplesso(a, b);
    NumeroComplesso z2 = new NumeroComplesso(c, d);
    // chiama metodo pubblico della classe NumeroComplesso
    NumeroComplesso ris = z1.prodotto(z2);
    modulo = ris.modulo();
    fase = ris.fase();
    lblRisultato.Text = string.Format("z1 * z2 = {0:f3} + j{1:f3}", modulo * Math.Cos(fase), modulo
                        * Math.Sin(fase));
}
else if (radDivisione.Checked)
{
    // acquisizione dati
    a = Convert.ToDouble(txtReal1.Text);
    b = Convert.ToDouble(txtImm1.Text);
    c = Convert.ToDouble(txtReal2.Text);
    d = Convert.ToDouble(txtImm2.Text);
    /* dichiara oggetto della classe NumeroComplesso
     e richiama il metodo costruttore con parametri */
    NumeroComplesso z1 = new NumeroComplesso(a, b);
    NumeroComplesso z2 = new NumeroComplesso(c, d);
    // chiama metodo pubblico della classe NumeroComplesso
    NumeroComplesso ris = z1.divisione(z2);
    modulo = ris.modulo();
    fase = ris.fase();
    lblRisultato.Text = string.Format("z1 / z2 = {0:f3} + j{1:f3}", modulo * Math.Cos(fase), modulo *
                        Math.Sin(fase));
}
else
{
    // acquisizione dati
    a = Convert.ToDouble(txtReal1.Text);
    b = Convert.ToDouble(txtImm1.Text);
    /* dichiara oggetto della classe NumeroComplesso
    e richiama il metodo costruttore con parametri */
    NumeroComplesso z1 = new NumeroComplesso(a, b);
    // chiama metodi pubblici della classe NumeroComplesso
    lblRisultato.Text = string.Format("|z1| = {0:f3}  fase z1 = {1:f3}°", z1.modulo(), z1.fase() * 180 /
                        Math.PI);
```

```
        }
    }

    private void Form1_Load(object sender, EventArgs e)
    {
        // seleziona radModFase
        radModFase.Checked = true;
    }
}

// Classe NumeroComplesso
class NumeroComplesso
{
    // dati della classe
    double real, imm;

    // metodi della classe
    public double modulo()
    {
        // calcolo del modulo del numero complesso
        double mod = Math.Pow(real, 2) + Math.Pow(imm, 2);
        return Math.Sqrt(mod);
    }

    public double fase()
    {
        // calcolo della fase in radianti del numero complesso
        return Math.Atan2(imm, real);
    }

    // metodo costruttore
    public NumeroComplesso(double cReal, double cImm)
    {
        real = cReal;
        imm = cImm;
    }

    public NumeroComplesso somma(NumeroComplesso z)
    {
        this.real = this.real + z.real;
        this.imm = this.imm + z.imm;
        return this;
    }

    public NumeroComplesso sottrazione(NumeroComplesso z)
    {
        this.real = this.real - z.real;
        this.imm = this.imm - z.imm;
        return this;
    }

    public NumeroComplesso prodotto(NumeroComplesso z)
    {
```

```
        NumeroComplesso ris = new NumeroComplesso(0, 0);
        ris.real = this.real * z.real - this.imm * z.imm;
        ris.imm = this.real * z.imm + this.imm * z.real;
        return ris;
      }

      public NumeroComplesso divisione(NumeroComplesso z)
      {
        NumeroComplesso ris = new NumeroComplesso(0, 0);
        ris.real = (this.real * z.real + this.imm * z.imm) / (Math.Pow(z.real, 2) + Math.Pow(z.imm, 2));
        ris.imm = (this.imm * z.real - this.real * z.imm) / (Math.Pow(z.real, 2) + Math.Pow(z.imm, 2));
        return ris;
      }
    }
}
```

Metodi statici

Un metodo statico è richiamabile specificando la classe a cui appartiene senza dover dichiarare un oggetto della classe.
Un esempio di metodo statico è il metodo **WriteLine** della classe **`Console`**, per chiamare tale metodo non si istanzia un oggetto della classe **`Console`**, ma si scrive semplicemente l'istruzione:

```
Console.WriteLine
```

Per dichiarare un metodo statico, è necessario usare la parola chiave **static**.
Creiamo quindi una classe che abbia tre metodi statici
Riprendiamo l'ultima versione del progetto **SetResetBit** che utilizzava i tre metodi, appartenenti alla classe **Form1**, **read**, **set** e **reset**: vogliamo togliere tali metodi dalla classe **Form1**, renderli statici ed inserirli in una classe di nome **Bit**.
La classe **Bit** verrà inoltre salvata in un file **.cs** che potremo aggiungere a qualsiasi progetto che richied l'utilizzo dei metodi di tale classe.
Dopo aver aperto il progetto **SetResetBit**, selezionare menu **Progetto | Aggiungi nuovo elemento…**e nella finestra di dialogo omonima selezionare l'opzione **Classe**, quindi assegnare, nella parte in basso della finestra, il nome **Bit** che diventerà quello della classe ed anche del file **.cs** in cui scriveremo il codice dei tre metodi statici.
Esaminando la classe **Bit** si osserva che il dei tre metodi non cambia, ma è sufficiente inserire nell'intestazione dei tre metodi la parola chiave **static** per renderli statici.
Passando all'analisi dei metodi presenti nella classe **Form1** si possono osservare le chiamate dei metodi statici **read**, **set** e **reset** della classe **Bit**: in ogni chiamata il nome del metodo è preceduto dal punto e dal nome della classe (Bit.read, Bit.set, Bit.reset).

```
// file Bit.cs
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SetResetBit
{
  class Bit
  {
    public static byte read(byte dato, byte numbit)
```

```
        {
            /* metodo che restituisce il valore logico di uno dei bit
             * (numerati da 0 a 15) di una variabile byte */
            return (byte)((dato & (1 << numbit)) >> numbit);
        }

        public static byte set(byte dato, byte numbit)
        {
            /* metodo che setta a 1 logico uno dei bit
             * (numerati da 0 a 15) di una variabile byte */
            return (byte)(dato | (1 << numbit));
        }

        public static byte reset(byte dato, byte numbit)
        {
            /* metodo che forza a 0 logico uno dei bit
             * (numerati da 0 a 15) di una variabile byte */
            return (byte)(dato & ~(1 << numbit));
        }
    }
}

namespace SetResetBit
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        // variabili globali
        byte numero = 0;
        /* metodo binario per conversione intero tra 0 e 255 in binario
         * utilizzante il metodo read   */
        private string binario(byte num)
        {
            string numString = "";
            for (byte i = 0; i <= 7; i++)
                numString = Bit.read(num, i) + numString;
            /* fa si che il metodo binario restituisca in uscita il
             * contenuto della variabile numString */
            return numString;
        }

        private void chkD7_CheckedChanged(object sender, EventArgs e)
        {
            if (chkD7.Checked)
                numero = Bit.set(numero, 7);
            else
                numero = Bit.reset(numero, 7);
            // chiama metodo binario
            lblNumero.Text = binario(numero);
```

```
}

private void chkD0_CheckedChanged(object sender, EventArgs e)
{
   if (chkD0.Checked)
      numero = Bit.set(numero, 0);
   else
      numero = Bit.reset(numero, 0);
   // chiama metodo binario
   lblNumero.Text = binario(numero);
}

private void chkD1_CheckedChanged(object sender, EventArgs e)
{
   if (chkD1.Checked)
      numero = Bit.set(numero, 1);
   else
      numero = Bit.reset(numero, 1);
   // chiama metodo binario
   lblNumero.Text = binario(numero);
}

private void chkD2_CheckedChanged(object sender, EventArgs e)
{
   if (chkD2.Checked)
      numero = Bit.set(numero, 2);
   else
      numero = Bit.reset(numero, 2);
   // chiama metodo binario
   lblNumero.Text = binario(numero);
}

private void chkD3_CheckedChanged(object sender, EventArgs e)
{
   if (chkD3.Checked)
      numero = Bit.set(numero, 3);
   else
      numero = Bit.reset(numero, 3);
   // chiama metodo binario
   lblNumero.Text = binario(numero);
}

private void chkD4_CheckedChanged(object sender, EventArgs e)
{
   if (chkD4.Checked)
      numero = Bit.set(numero, 4);
   else
      numero = Bit.reset(numero, 4);
   // chiama metodo binario
   lblNumero.Text = binario(numero);
}

private void chkD5_CheckedChanged(object sender, EventArgs e)
```

```
        {
            if (chkD5.Checked)
                numero = Bit.set(numero, 5);
            else
                numero = Bit.reset(numero, 5);
            // chiama metodo binario
            lblNumero.Text = binario(numero);
        }

        private void chkD6_CheckedChanged(object sender, EventArgs e)
        {
            if (chkD6.Checked)
                numero = Bit.set(numero, 6);
            else
                numero = Bit.reset(numero, 6);
            // chiama metodo binario
            lblNumero.Text = binario(numero);
        }
    }
}
```

Lettura e scrittura di file di testo

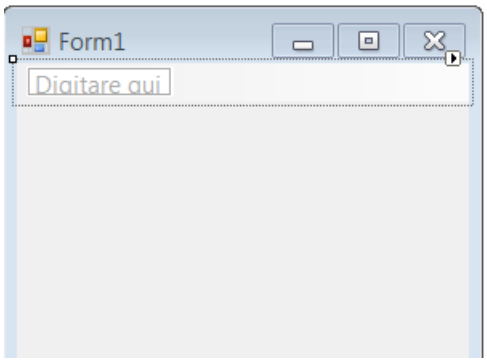
Le classi necessarie per leggere e scrivere file di testo sono contenute nel namespace System.IO che dovrà essere aggiunto, mediante la direttiva **using**, agli altri namespace del progetto.
Le classi in questione sono **StreamReader** e **StreamWriter.**

Creiamo dunque un progetto di nome **FileTesto** che permetta di salvare in un file di testo l'elenco alfabetico di un gruppo di persone digitato in un TextBox e di leggere il contenuto di tale file visualizzando l'elenco dei nomi in modo casuale in un ListBox.
Aggiungiamo al Form un TextBox, unListBox, un controllo **MenuStrip**, un controllo **SaveFileDialog** e un controllo **OpenFileDialog**.

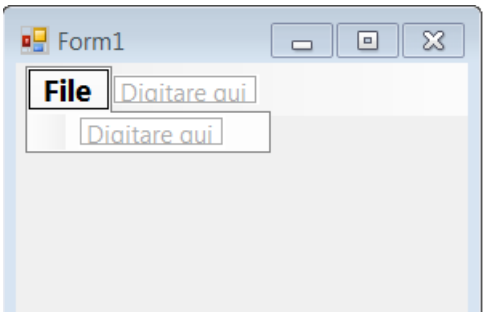
Proprietà Name:
TextBox: txtNomi
ListBox: lstNomi

Non è necessario modificare le proprietà Name degli altri tre controlli.

Il controllo **MenuStrip** permette di inserire dei menu nella parte superiore del Form; nel nostro caso, come si può osservare in figura, l'unico menu sarà il menu File con tre voci: **Salva con nome…**, **Apri e genera elenco casuale…** e **Cancella elenco alfabetico**. Una volta aggiunto il controllo **MenuStrip**, cliccare sul testo "Digitare qui" e scrivere **File**.
Scrivendo il nome della voce di menu si imposta la sua proprietà Text, mentre non è necessario modificare la proprietà Name.
Dopo l'aggiunta della voce di menu **File** le opzioni "Digitare qui" si trovano sotto e a destra della voce di menu **File**.

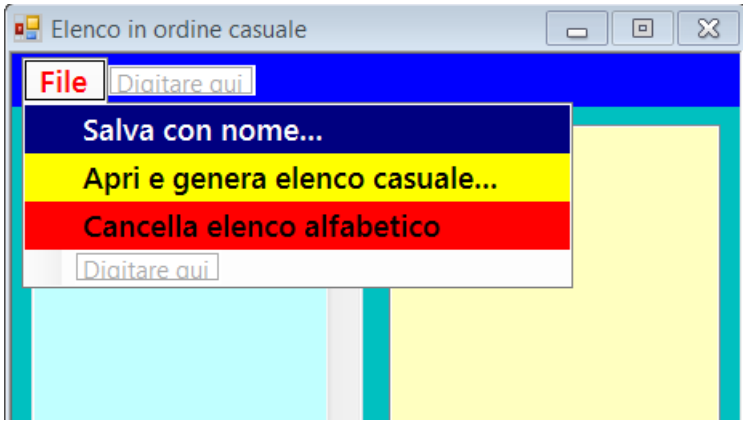


Aggiungiamo la voce di menu **Salva con nome…**sotto la voce di menu **File** e modifichiamone la proprietà Name in **mnuSalva**.
Con lo stesso procedimento aggiungiamo le altre due voci sotto la voce **Salva con nome…**assegnando loro le proprietà Name **mnuApri** e **mnuSvuota**.
Facendo doppio click in fase di progettazione sulle tre voci di menu contenute nel menu principale **File** creiamo i tre metodi gestori di evento.
Il menu ***mnuSvuota_Click*** conterrà un'unica istruzione che provvede a svuotare il TextBox.



Analizziamo ora il codice all'interno del metodo ***mnuSalva_Click***.
La prima istruzione dichiara un oggetto di nome ***File*** della classe **StreamWriter**, classe che supporta la scrittura di file di testo.
Il controllo ***saveFileDialog1*** permette l'utilizzo della finestra di dialogo ***Salva con nome*** con cui stabilire nome e path (percorso) del file in cui memorizzare l'elenco alfabetico digitato nel TextBox ***txtNomi***. L'istruzione:

```
saveFileDialog1.InitialDirectory = Application.StartupPath;
```



permette di impostare la cartella Release del progetto come cartella iniziale selezionata nella finestra di dialogo Salva con nome.
Le due istruzioni successive determinano, rispettivamente, il filtro usato per la casella combinata Tipo di File e quello selezionato di default.
L'istruzione:

```
if (saveFileDialog1.ShowDialog() ==
                    DialogResult.OK)
```

mostra la finestra di dialogo ***Salva con Nome*** chiamando il metodo ***ShowDialog()*** dell'oggetto ***saveFileDialog1***, quindi verifica se la finestra è stata chiusa cliccando sul pulsante OK.
Se la condizione all'interno dell' **if** è vera, l'oggetto **File** della classe **StreamWriter** viene inizializzato dal metodo costruttore della classe, che riceve come argomento in ingresso la stringa memorizzata nella proprietà ***FileName*** dell'oggetto ***saveFileDialog1***; tale stringa contiene il nome del file digitato nell'omonima casella di testo della finestra di dialogo.
Con l'istruzione:

```
File.WriteLine(txtNomi.Text);
```

scriviamo nel file di testo creato o selezionato in precedenza il contenuto del TextBox.
L'istruzione:

```
File.Close();
```

chiude il file per la scrittura.
Analizziamo ora il codice del metodo ***mnuApri_Click***.
L'istruzione:

```
Random num = new Random();
```

dichiara l'oggetto ***num*** della classe **Random** e lo inizializza con il metodo costruttore della classe.
L'oggetto ***num*** ci permetterà di generare dei numeri interi casuali che useremo per generare un elenco casuale dei nomi digitati.
L'istruzione:

```
StreamReader File;
```

dichiara un oggetto di nome ***File*** della classe **StreamReader**, classe che supporta la lettura di file di testo. Dopo aver rimosso tutti gli elementi dal ListBox, si dichiara una matrice **string** a due dimensioni con 35 righe (numero massimo di persone) e due colonne: la prima colonna conterrà i nomi delle persone acquisiti dal TextBox, la seconda colonna, inizializzata con delle stringhe vuote, conterrà degli asterischi in corrispondenza dei nomi estratti in modo casuale mediante l'oggetto ***num*** della classe **Random**.
Le istruzioni successive, analoghe a quelle viste in precedenza relativamente al controllo ***saveFileDialog1***, operano sul controllo ***OpenFileDialog*** che permette l'utilizzo della finestra di dialogo ***Apri*** con cui sfogliare le cartelle per selezionare il file contenente l'elenco alfabetico di persone da disporre in ordine casuale.
Anche in questo caso, se si chiude la finestra di dialogo ***Apri*** cliccando sul pulsante OK, l'oggetto ***File*** della classe **StreamReader** verrà inizializzato dal metodo costruttore della classe, che riceve come argomento in ingresso la stringa memorizzata nella proprietà ***FileName*** dell'oggetto.
A questo punto è possibile leggere il contenuto del file selezionato una riga alla volta, dall'alto verso il basso, mediante le istruzioni presenti nel ciclo **while**.
Per prima cosa si verifica la condizione all'interno del **while** testando la proprietà booleana ***EndOfStream*** dell'oggetto ***File***: tale proprietà è **true** se la lettura del file è stata completata, **false** se vi sono ancora righe del file da leggere. Negando con l'operatore NOT (!) la proprietà ***EndOfStream*** facciamo in modo che le istruzioni nel ciclo **while** vengano eseguite quando ***EndOfStream*** assume il valore **false**, cioè quando il file contiene righe non ancora lette.
L'istruzione:

```
nomi[i, 0] = File.ReadLine();
```

leggerà una riga del file, cioè un nome per riga, salvandolo nella prima colonna dell'i-esima riga della matrice, con ***i*** variabile contatore che dovrà, al termine del ciclo while, contenere il numero di nomi memorizzati nella matrice. Le ultime due istruzioni all'interno del **while** inizializzano gli elementi della seconda colonna della matrice con stringhe vuote e incrementano la variabile contatore ***i***.
Terminata la lettura del file si chiude il file per la lettura.
Le istruzioni all'interno del secondo ciclo **while** prelevano, in modo casuale, un elemento alla volta della matrice ***nomi*** visualizzandolo nel ListBox.
L'istruzione:

```
numEstr = num.Next(0,i);
```

genera un intero casuale maggiore o uguale a zero e minore del numero presente nella variabile contatore ***i***; tale numero rappresenterà quindi il numero di riga da cui prelevare il nome della persona estratta, se non già estratta precedentemente. Con l'istruzione:

```
if (nomi[numEstr, 1] == "")
```

si legge il contenuto della matrice ***nomi*** in corrispondenza seconda colonna e della riga avente come indice il numero generato casualmente: se l'elemento memorizzato è una stringa vuota significa che il nome corrispondente non è ancora stato estratto; in questo caso si aggiunge al ListBox il nome corrispondente preceduto da un numero progressivo che si ottiene aumentando di uno la variabile contatore ***k*** che memorizza il numero di estrazioni.
Si segna infine il nome in questione come già letto scrivendo nella seconda colonna un asterisco e si incrementa ***k*** di uno.
L'istruzione while (k < i) garantisce un numero di estrazioni pari al numero di persone memorizzate nel file di testo.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
```

```
using System.Windows.Forms;
// Classi StreamWriter e StreamReader
using System.IO;

namespace FileTesto
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
      }

      private void mnuSalva_Click(object sender, EventArgs e)
      {
         // Dichiara oggeto classe StreamWriter per scrittura file di testo
         StreamWriter File;
         /* Imposta cartella Release del progetto come cartella iniziale
            selezionata nella finestra di dialogo Salva con nome */
         saveFileDialog1.InitialDirectory = Application.StartupPath;
         // stabilisce il filtro usato per la casella combinata Tipo di File
         saveFileDialog1.Filter = "file di testo (*.txt)|*.txt|"
            + "tutti i file (*.*)|*.*";
         // determina il filtro selezionato di default
         saveFileDialog1.FilterIndex = 1;
         /* Mostra finestra di dialogo Salva con Nome e verifica se si è chiusa
          * cliccando sul pulsante OK */
         if (saveFileDialog1.ShowDialog() == DialogResult.OK)
         {
            /* Chiama il costruttore della classe StreamWriter passandogli
             * come argomento una stringa contenente il nome del file digitato
             * nella finestra di dialogo Salva con nome */
            File = new StreamWriter(saveFileDialog1.FileName);
            // scrive il contenuto del TextBox nel file
            File.WriteLine(txtNomi.Text);
            // chiude il file di testo
            File.Close();
         }
      }

    private void mnuApri_Click(object sender, EventArgs e)
      {
         int i = 0, k = 0;
         int numEstr;
         // dichiara un oggetto della classe Random
         Random num = new Random();
         // Dichiara oggeto classe StreamReader per lettura file di testo
         StreamReader File;
         // Rimuove tutti gli elementi dal ListBox
         lstNomi.Items.Clear();

        /* dichiara matrice a due dimensioni per nomi persone
         * (prima colonna) e marcatori estrazione (seconda colonna) */
```

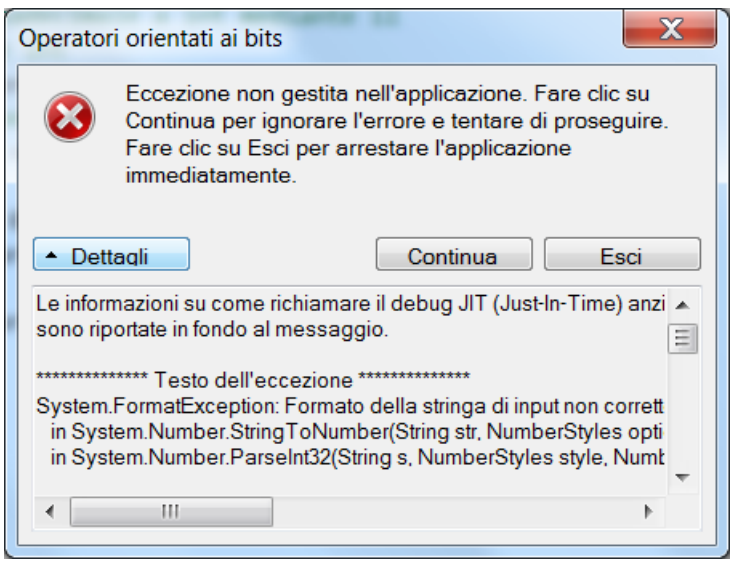
```
    string [,] nomi = new string[35, 2];
    /* Imposta cartella Release del progetto come cartella iniziale
       selezionata nella finestra di dialogo Salva con nome */
    openFileDialog1.InitialDirectory = Application.StartupPath;
    // svuota TextBox nella finestra di dialogo contenente nome file
    openFileDialog1.FileName = "";
    // stabilisce il filtro usato per il ComboBox Tipo di File
    openFileDialog1.Filter = "file di testo (*.txt)|*.txt|"
       + "tutti i file (*.*)|*.*";
    // determina il filtro selezionato di default
    openFileDialog1.FilterIndex = 1;
    /* Mostra finestra di dialogo Apri e verifica se si è chiusa
     * cliccando sul pulsante OK */
    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    {
       /* Chiama il costruttore della classe StreamReader passandogli
        * come argomento una stringa contenente il nome del file
        * selezionato nella finestra di dialogo Salva con nome */
       File = new StreamReader(openFileDialog1.FileName);
       // esegue se non si è arrivati all'ultima riga del file
       while (!File.EndOfStream)
       {
          /* legge nomi dal file una riga per volta caricandoli nella
           * prima colonna di nomi[]          */
          nomi[i, 0] = File.ReadLine();
          // segna tutti i nomi come non estratti(stringa vuota)
          nomi[i, 1] = "";
          i++; // incrementa contatore numero persone
       }
       // chiude stream di lettura
       File.Close();
       /* esegue se il numero di estrazioni è minore
        * del numero di persone */
       while (k < i)
       {
          /* estrae casualmente un intero nell'intervallo tra
           * 0 e i-1, con i = numero persone */
          numEstr = num.Next(0,i);
          /* verifica se l'intero estratto corrisponde a
           * un nome non ancora estratto */
          if (nomi[numEstr, 1] == "")
          {
             // aggiunge al ListBox il nome estratto
             lstNomi.Items.Add((k + 1) + " " + nomi[numEstr, 0]);
             /* segna il nome come estratto inserendo un asterisco nella
              * seconda colonna */
             nomi[numEstr, 1] = "*";
             k++; // incrementa contatore estrazioni
          }
       }
    }
}
```

```
        private void mnuSvuota_Click(object sender, EventArgs e)
        {
            // svuota TextBox
            txtNomi.Text = "";
        }
    }
}
```

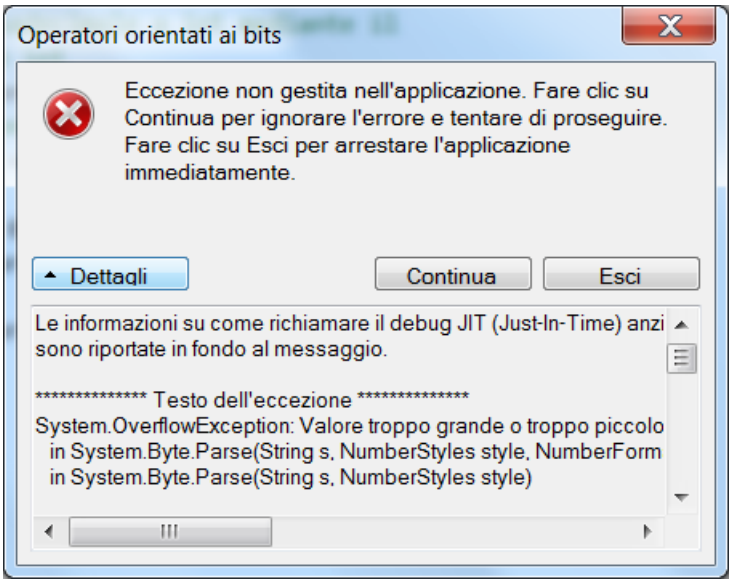
Gestione delle eccezioni

Un'eccezione è un problema che si verifica durante l'esecuzione del un programma e che deve essere risolto prima che il programma possa procedere. L'eccezione può essere causata da codice errato, errori dell'utente o circostanze che esulano dal controllo del programmatore o dell'utente dell'applicazione.
Tra gli esempi di eccezioni si trovano l'impossibilità di aprire un file perché non è individuabile, perché l'utente non ha inserito il drive USB contenente il file o perché il file è danneggiato.
Come esempio di eccezione causata da un errore dell'utente, riprendiamo il progetto **OperatoriBitWise** in cui si operava la conversione decimale-binario mediante l'operatore And (&).
Modifichiamo il codice dichiarando di tipo **byte** la variabile in cui salvare il numero esadecimale intero da convertire in binario. L'intervallo di valori che si possono convertire andrà quindi da 0 ad FF.
Il codice diventerà quindi il seguente:

```
private void btnBinario_Click(object sender, EventArgs e)
        {
            string numeroString = txtEsadec.Text;
            string binario = "";
            byte numeroByte = 0;
            // conversione da stringa esadecimale a int mediante il
            // metodo Parse della classe int
            numeroByte = byte.Parse(numeroString, NumberStyles.HexNumber);
            // conversione in binario con operatore & (AND)
            for (int i = 1; i <= 128; i = i*2)
            {
                if ((numeroByte & i) == 0)
                    binario = '0' + binario;
                else
                    binario = '1' + binario;
            }
            lblBinario.Text = binario;
        }
```



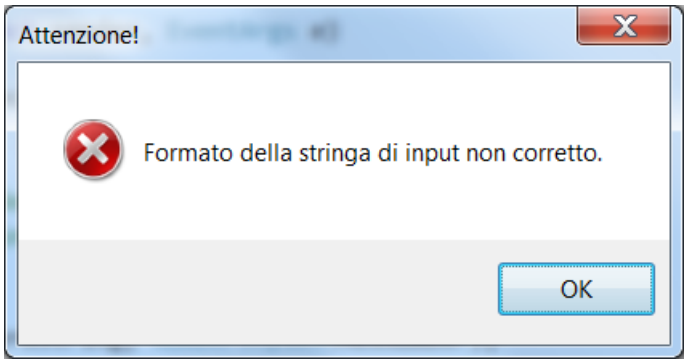
Se l'utente, in fase di esecuzione, clicca sul pulsante che avvia la conversione senza aver digitato nulla nel TextBox, l'applicazione si blocca e, se si clicca sul pulsante **Dettagli**, viene visualizzata la finestra di messaggio mostrata a fianco, con il messaggio "Formato della stringa di input non corretto".
Dal messaggio riportato si deduce che l'informazione sull'errore verificatosi è memorizzata nella classe **FormatException**.



Un altro tipo di eccezione viene sollevata se l'utente digita un numero esadecimale maggiore di FF; in questo caso la finestra di messaggio mostrata a fianco ci informerà che l'informazione sull'errore verificatosi è memorizzata nella classe **OverflowException**.
Queste due classi ci permetteranno di modificare il codice in modo da gestire questi due tipi di eccezioni in modo che il programma non si blocchi.
Quando si richiama un blocco di codice che potrebbe sollevare eccezioni, è possibile racchiuderlo nell'istruzione di controllo **try** per intercettare eventuali eccezioni sollevate in quel codice.
Nel blocco tra {} che segue il **try** viene scritto il codice soggetto a controllo, nel nostro caso l'istruzione:

```
numeroByte = byte.Parse(numeroString, NumberStyles.HexNumber);
```

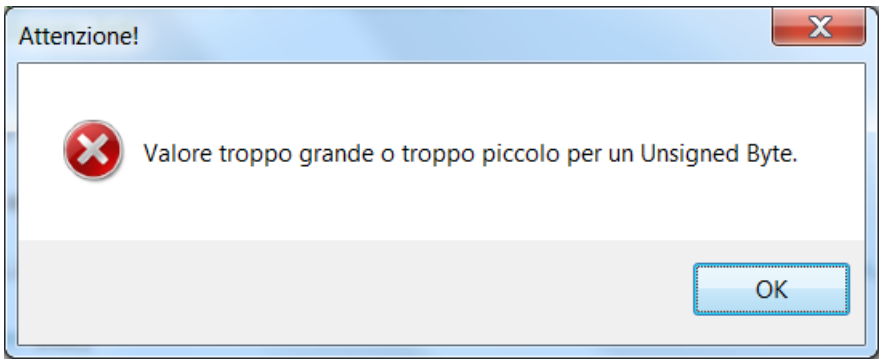
I tipi di eccezioni che si vogliono intercettare sono invece indicate nelle istruzioni **catch**: ne esiste una per ciascuna eccezione da gestire. Nel nostro caso, volendo gestire due tipi di eccezioni, avremo due istruzioni **catch**: una relativa alle **FormatException**,l'altra alle **OverflowException**. Se, nel corso dell'esecuzione del codice, nel blocco **try** verrà sollevata una delle due eccezioni, questa verrà intercettata dalla relativa istruzione **catch**.
In entrambi i casi l'applicazione, se l'eccezione viene sollevata, invece di bloccarsi visualizzerà un **MessageBox** (finestra di messaggio) con un messaggio di errore: chiusa la finestra di messaggio l'applicazione rimarrà comunque in esecuzione.



Ad esempio, l'istruzione:

```
MessageBox.Show(ex1.Message,"Attenzione!",
MessageBoxButtons.OK, MessageBoxIcon.Error);
```

mostrerà il **MessageBox** riportato a fianco.
Si osservi come il messaggio relativo al tipo di errore sia contenuto nella proprietà **Message** dell'oggetto **ex1**; oggetto della classe **FormatException** dichiarato all'interno del primo **catch**.



Una seconda istruzione MessageBox.Show, inserita all'interno del secondo **catch,** mostrerà la finestra di messaggio fianco quando verrà sollevata un'eccezione gestita dalla classe **OverflowException**.
In questo caso l'informazione sull'errore verificatosi
sarà contenuto nella proprietà **Message** dell'oggetto **ex2**; oggetto della classe **OverflowException** dichiarato all'interno del secondo **catch**.
Il codice finale sarà quindi il seguente:

```
private void btnBinario_Click(object sender, EventArgs e)
 {
    string numeroString = txtEsadec.Text;
    string binario = "";
    byte numeroByte = 0;
    // conversione da stringa esadecimale a int mediante il
    // metodo Parse della classe int
    try
    {
       numeroByte = byte.Parse(numeroString, NumberStyles.HexNumber);
    }
    catch (FormatException ex1)
    {
       MessageBox.Show(ex1.Message,"Attenzione!", MessageBoxButtons.OK,
                                MessageBoxIcon.Error);
    }
    catch (OverflowException ex2)
    {
       MessageBox.Show(ex2.Message, "Attenzione!", MessageBoxButtons.OK,
                                MessageBoxIcon.Error);
    }
    // conversione in binario con operatore & (AND)
    for (int i = 1; i <= 128; i = i*2)
    {
       if ((numeroByte & i) == 0)
          binario = '0' + binario;
       else
          binario = '1' + binario;
    }
    lblBinario.Text = binario;
 }
```

Interfacce seriali

L'interfaccia seriale RS-232

L'interfaccia seriale RS232 è uno standard nato nei primi anni '60 costituito da una serie di protocolli meccanici, elettrici ed informatici che rendono possibile lo scambio di informazioni a bassa velocità tra dispositivi digitali. Tale standard si è evoluto nel corso degli anni pur mantenendosi in larga parte invariato.
Pur essendo un protocollo piuttosto vecchio, attualmente ***la RS-232 è ancora largamente utilizzata per la comunicazione a bassa velocità tra microcontrollori, dispositivi industriali ed altri circuiti relativamente semplici che non necessitano di particolare velocità***; è invece praticamente scomparsa nel campo delle telecomunicazioni.

Caratteristiche generali delle interfacce seriali

L'interfaccia RS-232 utilizza un protocollo seriale asincrono ed il collegamento è di tipo punto-punto.

- **Seriale** specifica che i bit che costituiscono l'informazione sono trasmessi sequenzialmente, uno alla volta su di un solo "filo". Questo termine è in genere contrapposto a "parallelo", termine che indica una tipologia di trasmissione in cui i dati viaggiano contemporaneamente su più fili, per esempio 8, 16 o 32. Esempi di trasmissioni seriali sono le trasmissioni in una fibra ottica, in un cavo ethernet, USB o FireWire, in un bus PCI-Express.

- **Asincrono** significa**,** in questo contesto, che i dati sono trasmessi senza l'aggiunta di un segnale di clock, cioè senza alcun segnale comune tra trasmettitore e ricevitore destinato a sincronizzare il flusso di informazioni; ovviamente sia il trasmettitore che il ricevitore devono comunque essere dotati di un proprio clock locale per poter interpretare correttamente i dati

- Una trasmissione è di tipo **punto-punto** quando nella comunicazione è presente, per ciascun segnale utilizzato, un solo trasmettitore ed un solo ricevitore.

La velocità di trasmissione

Le unità di misura della velocità di trasmissione sono essenzialmente due: il ***baud*** ed il ***bit per secondo*** (bps o bit/s), spesso trattate erroneamente come sinonimi.

Il ***bps*** indica, come dice il nome, quanti bit al secondo sono trasmessi lungo la linea. Questa è la velocità effettiva della trasmissione vista dai dispositivi digitali. Nel caso di trasmissione a due livelli (cioè è presente un livello di tensione alto ed uno basso) **baud** rate e bps coincidono numericamente, da cui la *parziale* equivalenza dei due termini. Nel caso di trasmissioni a più livelli, invece, è possibile trasmettere con un solo livello più bit, ottenendo un baud rate minore a parità di informazioni trasmesse.

Lo standard RS232 utilizza due livelli quindi il baud rate coincide numericamente con il bps.
Nei normali PC le velocità di trasmissione consentite con lo standard RS232 sono (in **bps**):

50 – 300 – 600 – 2400 – 4800 – 9600 – 19200 – 38400 – 57600 - 115200

Trasmissioni half-duplex e full-duplex

I due termini fanno riferimento alla situazione in cui due dispositivi si scambiano informazioni tra di loro, comportandosi entrambi sia da trasmettitori (**Tx**) sia da ricevitori (**Rx**).

Half-duplex indica che la trasmissione è bidirezionale ma non contemporanea nei due versi: in un determinato istante uno solo dei due dispositivi emette segnali, l'altro ascolta. Quando è necessario, si

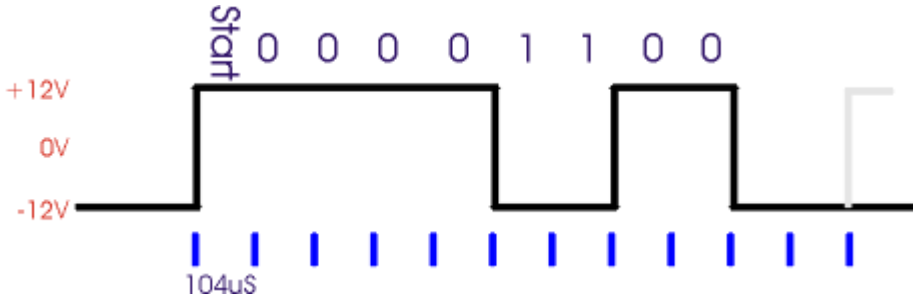
scambiano di ruolo. **Full-duplex** indica che la trasmissione è bidirezionale e contemporanea. In questo caso sono necessari due fili, uno per ciascun verso di trasmissione. Se la trasmissione è sempre in un solo verso, si parla di **simplex**. Lo standard RS232 permette tutte e tre queste modalità di funzionamento in quanto è utilizzato un conduttore separato per ciascun verso di trasmissione.

Parametri elettrici della RS-232

La tensione di uscita da un trasmettitore RS232 deve essere compresa in valore assoluto tra 5V e 25V (quest'ultimo valore ridotto a 13V in alcune revisioni dello standard). A volte le tensioni in uscita sono intenzionalmente diminuite a +/- 6V anziché 12V per permettere minori emissioni elettromagnetiche e favorire maggiori velocità di trasmissione.
Il ricevitore deve funzionare correttamente con tensioni di ingresso comprese, sempre in modulo, tra i 3V ed i 25V.
La corrente di corto circuito deve essere minore di 100mA.

Come è fatto un segnale RS-232

La cosa più semplice per descrivere un segnale RS232 è partire con un esempio.
Nell'immagine che segue è visualizzato, in modo idealizzato, cosa appare collegando un oscilloscopio ad un filo su cui transita un segnale RS-232 a 9600 bps del tipo 8n2 (più avanti verrà spiegata questa sigla) rappresentante il valore binario 00110000.



L'ampiezza del segnale è caratterizzata da un valore "alto" pari a circa +12V ed un valore "basso" pari a circa –12V. Da notare che, nello standard RS-232 un segnale alto rappresenta lo zero logico ed uno basso un uno logico.
A volte un segnale alto (+12V, cioè uno zero logico) è indicato come **space** ed uno basso (-12V, uno logico) come **mark**.
Tutte le transizioni appaiono in corrispondenza di multipli di 104 µs, pari ad 1/9600, cioè il tempo di bit è esattamente l'inverso del baud rate.
La linea si trova inizialmente nello stato di riposo, bassa (nessun dato in transito); la prima transizione da basso in alto indica l'inizio della trasmissione (inizia il "bit di start", lungo esattamente 104us). Segue il bit meno significativo (LSB), dopo altri 104 µs il secondo bit, e così via, per otto volte, fino al bit più significativo (MSB). Da notare che il byte è trasmesso "al contrario", cioè va letto da destra verso sinistra.
Segue infine un periodo di riposo della linea di ***almeno* 208 us**, cioè due bit di stop e quindi (eventualmente) inizia un nuovo pacchetto di bit con un nuovo bit di start (in grigio nel disegno).
E' anche possibile usare un solo bit di stop.
E' importante garantire il rigoroso rispetto della durata dei singoli bit: infatti non è presente alcun segnale di clock comune a trasmettitore e ricevitore e l'unico elemento di sincronizzazione è dato dal fronte si salita del bit di start. Come linea guida occorre considerare che il campionamento in ricezione è effettuato di norma al centro di ciascun bit.

Collegare una porta TTL o CMOS alla RS232

In genere i segnali utilizzati dai sistemi digitali sono "TTL compatibili", cioè variano tra 0 e 5V, oppure variano tra 0 V e 3,3 V: non sono quindi direttamente compatibili con la standard RS232. In commercio

esistono appositi ***traslatori di livello*** che hanno il compito di fornire sia in trasmissione che in ricezione gli opportuni livelli pur non modificando la forma del segnale trasmesso.
Il **MAX232** (ed integrati simili) è un circuito integrato che permette il collegamento tra logica TTL o CMOS a 5V e le tensioni RS-232, partendo da un'alimentazione a 5V.
Sono disponibili anche integrati che richiedono un'alimentazione di soli 3.3V (p.e. il **MAX3232**).
La sezione ricevente del MAX232 è costituita da due porte invertenti che accettano in ingresso una tensione di +/- 12V (o altra tensione compatibile allo standard RS232) ed in uscita presentano un segnale TTL compatibile.
La sezione trasmittente ha due driver invertenti con in ingresso TTL compatibile e capaci di erogare a vuoto una tensione di poco meno di +/- 10V, compatibile con lo standard RS232.

Piedinatura del connettore RS-232 del PC

Nei personal computer sono disponibili due tipi di connettori RS-232: DB9 (nove pin) e DB25 (25 pin, il connettore originale e presente solo sui PC più vecchi); ambedue i connettori sono maschi e praticamente identici dal punto di vista funzionale anche se non coincidente con quello proposto dallo standard ufficiale.

Nell'immagine sopra è visualizzato il **connettore maschio DB9** presente sul PC (in realtà oggi lo standard RS232 è relegato solo ad ambiti industriali ed è stato sostituito nei pc dall'USB), mentre nella tabella in basso sono indicati i nomi dei segnali, il numero dei pin e la direzione del segnale (O = uscita dal PC I = ingresso del PC).

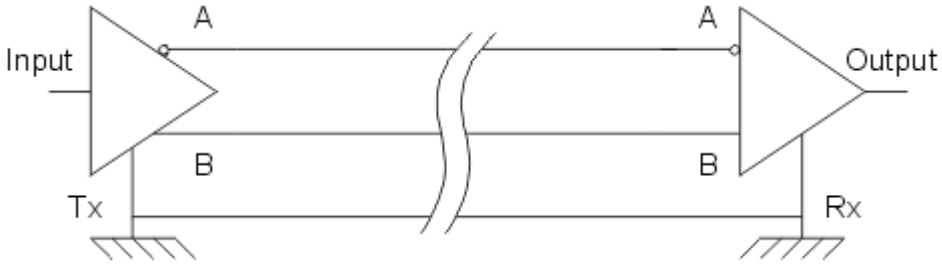
Sigla	Connettore maschio DB9	Input/Output	Nome
TD o TxD	3	Out	Dati trasmessi
RD o RxD	2	In	Dati ricevuti
RTS	7	Out	Request To Send
DTR	4	Out	Data Terminal Ready
CTS	8	In	Clear To Send
DSR	6	In	Data Set Ready
DCD	1	In	Data Carrier Detect

RI	9	In	Ring Indicator
SG	5	-	Terra

Standard RS422

Questo standard è stato originariamente proposto per la trasmissione di segnali digitali fino a 10 Mbit/s (10 milioni di bit al secondo) su distanze fino a 4000 piedi (circa 1200 m). Usando integrati moderni è inoltre possibile superare i limiti imposti dallo standard sia in termini di velocità che di distanza.

Lo standard RS422 prevede che ciascuna linea differenziale sia pilotata da un driver. I ricevitori possono essere fino a 10 ma è più comune l'utilizzo di questo standard nelle comunicazioni punto-punto, cioè per collegare un singolo trasmettitore (Tx) ad un singolo ricevitore (Rx), come rappresentato nello schema.



I due stati di ciascuna linea sono definiti nel seguente modo:

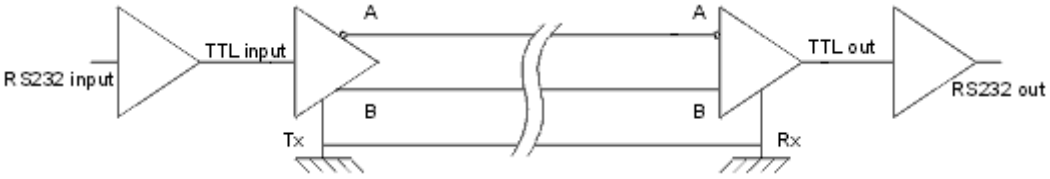
- Quando il terminale A è negativo rispetto a B, la linea rappresenta un uno binario. Tale stato rappresenta anche l'assenza di segnale (idle state)
- Quando il terminale A è positivo rispetto a B, la linea rappresenta uno zero binario

La tensione differenziale è quella che effettivamente trasmette l'informazione ed è positiva o negativa in funzione del livello logico trasmesso. All'uscita del trasmettitore la differenza di potenziale tra le linee A e B deve essere di almeno 4 V e la tensione di modo comune deve essere minore di 7 V (normalmente una linea vale circa 0 V e l'altra circa 5 V). Il ricevitore deve essere in grado di interpretare correttamente lo stato della linea quando la differenza di potenziale è superiore in modulo a 200 mV

Conversione da RS232 a RS422

Spesso i trasmettitori ed i ricevitori RS422 sono utilizzati per estendere la portata delle porte RS232, in genere limitata a pochi metri oppure poche decine se le velocità rimane sotto i 100 kb/s. Per fare ciò viene effettuata una semplice conversione dei livelli elettrici conservando per esempio la classica struttura del byte composto da un bit di start, da 6 a 8 bit di dati ed almeno un bit di stop .

Per trasmettere un singolo segnale è utilizzata una struttura simile a quella di seguito rappresentata: il segnale RS232 viene prima convertito in TTL e quindi in RS422; alla ricezione viene effettuata la conversione opposta. È necessario prevedere almeno due coppie di cavi, uno per ciascuna direzione, operazione facilitata dal fatto che ciascun circuito integrato contiene a volte sia il ricevitore che il trasmettitore.



Dal punto di vista del software, la connessione è perfettamente trasparente a condizione di usare un sufficiente numero di cavi e quindi sono utilizzabili tutti i protocolli normalmente adottati con le porte

RS232). Per il funzionamento di questo circuito è richiesta una alimentazione esterna, in genere di 5 volt. In commercio esistono schede **Bus Pci RS 422** da inserire sul bus PCI del PC che permettono di arrivare a distanze pari a circa 1300 m con velocità di trasferimento che possono arrivare ad alcuni Mbit/s.

Controllo Serialport in Visual C#

Il controllo SerialPort fornisce metodi per la lettura/scrittura di dati dal/sul port seriale RS232 e per la gestione dei segnali/pins di handshaking RTS, DTR, CTS, DSR e DCD.

Per quanto riguarda la lettura del port seriale (ricezione):

a) selezionare il controllo serialPort1
b) impostare le proprietà **PortName** (COM1, COM2,etc.) e **BaudRate**
c) selezionare l'evento DataReceived e scrivere:

```
private void serialPort1_DataReceived(…)
{
     int datoRx = serialPort1.ReadByte();
}
```

L'evento **`DataReceived`** si verifica ogni volta che il buffer di ricezione della porta seriale riceve un dato. Il metodo **`ReadByte()`** è di tipo **int** e legge un byte dal buffer di ricezione, eliminandolo dal buffer stesso. E' possibile, in alternativa a ReadByte(), usare il metodo **`Read()`**:

```
private void serialPort1_DataReceived(…)
{
    byte[] dato = new byte[1];
    serialPort1.Read(dato, 0, 1);
}
```

Il dato letto dal buffer di ricezione della porta seriale viene salvato nell'elemento numero zero della matrice dato[].
E'anche possibile fare in modo di attivare l'evento **`DataReceived`** dopo la ricezione, ad esempio, di due byte tramite la proprietà **`ReceivedBytesThreshold`**:

```
serialPort1.ReceivedBytesThreshold = 2;
```

Per leggere due bytes dal buffer di ricezione della porta seriale:

```
private void serialPort1_DataReceived(…)
{
    byte[] dato = new byte[2];
    serialPort1.Read(dato, 0, 2);
}
```

I due byte letti dal buffer di ricezione saranno salvati in `dato[0]` e `dato[1]`.

Per quanto riguarda la scrittura del port seriale (trasmissione) è possibile usare il metodo **`Write()`**:

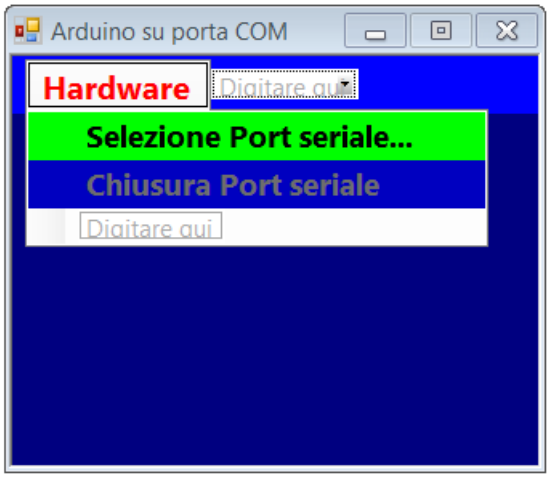
```
private void button1_Click(object sender, EventArgs e)
      {
          byte[] dato = new byte[1];
          dato[0] = 121;
          serialPort1.Write(dato, 0, 1);
}
```

Il metodo **`Write()`** scrive nel buffer di trasmissione della porta seriale il byte presente nell'elemento numero zero della matrice dato[].

E'possibile forzare a 0 logico (Space) e 1 logico(Mark) i pins di output DTR e RTS:

```
serialPort1.DtrEnable = true; // forza a Space
serialPort1.DtrEnable = false; // forza a Mark
serialPort1.RtsEnable = true; // forza a Space
serialPort1.RtsEnable = false; // forza a Mark
```

Analogamente, è possibile verificare lo stato dei pins di input CTS, DSR, CD mediante le rispettive proprietà.
Creiamo ora il primo progetto utilizzante un controllo **serialPort**, progetto cheutilizzeremo in seguito come base di partenza pertutte le applicazioni che gestiranno lo scambio di informazioni, tramite port RS232, tra PC ed altri dispositivi (ad esempio, display a Led, LCD, **scheda a microcontrollore Arduino**).
Il progetto che creeremo ora ha il solo compito di visualizzare tutti i port RS232 presenti nel PC e di selezionarne uno a piacere scegliendo la velocità di trasferimento in baud tra quelle consentite.



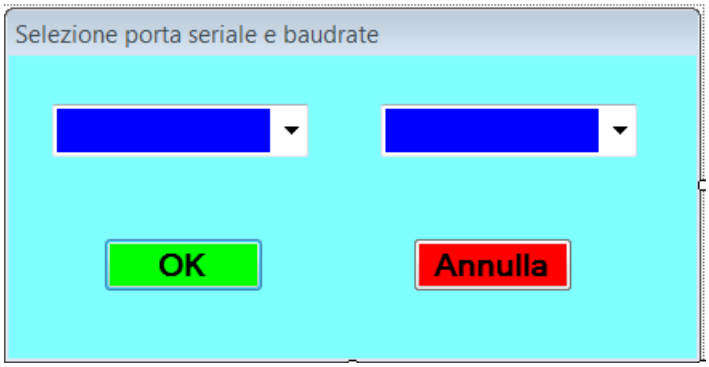
Creiamo quindi un nuovo progetto con il nome **PortRS232**.
Aggiungiamo al Form i controlli **serialPort** e **MenuStrip** che non è necessario rinominare.
Non è necessario modificare nessuna delle proprietà del controllo **serialPort1**, mentre occorre impostare la proprietà Text del controllo **menuStrip1** su **Hardware**, quindi creare sotto quest'ultima voce le voci di menu **Selezione Port seriale…** e **Chiusura Port seriale** con, rispettivamente, le proprietà *name* **mnuSelezione** e **mnuChiusura**.
Per eliminare il pulsante d'ingrandimento del Form impostiamo la sua proprietà **MaximizeBox** su **false** e per impedire il ridimensionamento del Form in fase di esecuzione impostiamo la sua proprietà **FormBorderStyle** su **Fixed3D.**
Impostiamo inoltre la proprietà ***Enabled*** di **mnuChiusura** su **false** per disabilitare il menu in questione all'avvio dell'applicazione.
Facciamo ora doppio click sulle voci di menu **Selezione Port seriale…** e **Chiusura Port seriale** per creare i metodi gestori di evento ***mnuSelezione_Click*** e ***mnuChiusura_Click***, quindi creiamo il metodo gestore d'evento ***Form1_FormClosed*** selezionando il Form1, andando nella Finestra proprietà, cliccando sull'icona  e facendo doppio click sull'evento *FormClosed*.

Ora aggiungiamo al progetto un secondo Form che dovrà diventare una finestra di dialogo in cui selezionare porta seriale e baudrate.

Creazione di un Form di dialogo

1) menu Progetto | Aggiungi Windows Form...
   selezionare Windows Form (non cambiare il nome Form2)
2) Aggiungere un primo pulsante a Form2 con proprietà:
   a) Name = btnOK
   b) Text = OK
   c) DialogResult = OK
   d) TabIndex = 0 (cursore nel ComboBox al caricamento del Form)
3) Aggiungere un secondo pulsante a Form2 con proprietà:
   a) Name = btnCancel
   b) Text = Annulla
   c) DialogResult = Cancel
   d) TabIndex = 1 (se si digita il Tab il cursore si sposta suquesto pulsante)

4) Aggiungere un ComboBox a Form2 con proprietà:
   a) Name = cmbPortName
   b) Modifiers = Internal (ComboBox in Form2 visibili anche dal codice presente in Form1)
5) Aggiungere un secondo ComboBox a Form2 con proprietà:
   a) Name = cmbBaudRate
   b) TabIndex = 1
   c) Modifiers = Internal



Impostare quindi le seguenti **proprietà del Form2**:

a) ControlBox = False (elimina pulsanti in alto a destra e sinistra)
b) Text = Selezione porta seriale e baudrate
c) FormBorderStyle = FixedDialog (comportamento di bordo e barra del titolo del form)
d) StartPosition = CenterParent (posizione iniziale del Form2 centrata rispetto a Form1)
e) AcceptButton = btnOK Tasto <INVIO> corrispondente a pulsante OK
f) CancelButton = btnCancel Tasto <ESC> corrispondente al pulsante Annulla

La classe **SerialPort** necessaria per ottenere le porte seriali presenti nel PC è contenuta nel namespace System.IO.Ports che dovrà essere aggiunto, mediante la direttiva **using**, agli altri namespace del progetto con la seguente istruzione:

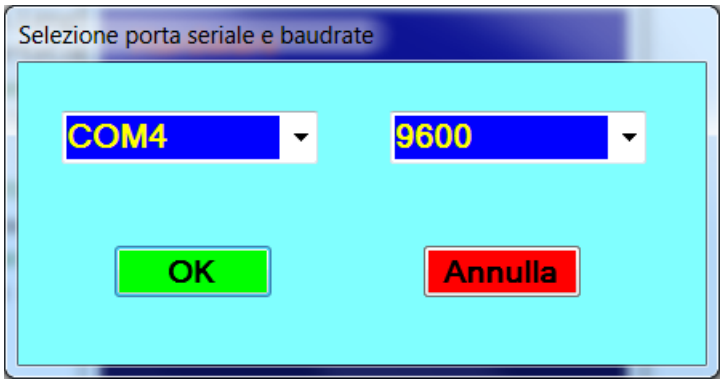
```
using System.IO.Ports;
```

Creiamo ora il metodo gestore d'evento **Form2_Load** facendo doppio click su una zona libera del Form2 e scriviamo il codice seguente:

```
private void Form2_Load(object sender, EventArgs e)
    {
      // carica i valori del BaudRate nel primo ComboBox
      cmbBaudRate.Items.Add(300);
      cmbBaudRate.Items.Add(1200);
      cmbBaudRate.Items.Add(2400);
      cmbBaudRate.Items.Add(4800);
      cmbBaudRate.Items.Add(9600);
      cmbBaudRate.Items.Add(14400);
      cmbBaudRate.Items.Add(19200);
      cmbBaudRate.Items.Add(28800);
      cmbBaudRate.Items.Add(38400);
      cmbBaudRate.Items.Add(57600);
      cmbBaudRate.Items.Add(115200);
      // fissa indice dell'elemento selezionato di default
      cmbBaudRate.SelectedIndex = 4;
      // fornisce un elenco delle porte seriali
      string[] porteCom = SerialPort.GetPortNames();
      // visualizza porte COM nel secondo ComboBox
      for (int i = 0; i < porteCom.Length; i++)
         cmbPortName.Items.Add(porteCom[i]);
       // verifica presenza di almeno una porta COM
      if (cmbPortName.Items.Count != 0)
         // imposta indice elemento selezionato di default
         cmbPortName.SelectedIndex = cmbPortName.Items.Count - 1;
      else
```

```
        {

            MessageBox.Show("Nessuna porta COM rilevata", "Attenzione!", MessageBoxButtons.OK,
                                MessageBoxIcon.Information);
            // chiude il Form2
            this.Close();
        }
    }
```

Analizziamo il codice precedente. Le prime undici istruzioni caricano nel secondo ComboBox gli undici valori consentiti per il baudrate.



L'istruzione:

```
cmbBaudRate.SelectedIndex = 4;
```

imposta come velocità visualizzata di default la quinta (cioè l'elemento numero quattro del ComboBox) corrispondente a 9600 baud.
L'istruzione:

```
string[] porteCom = SerialPort.GetPortNames();
```

chiama il metodo statico *GetPortNames* della classe **SerialPort** che restituisce na matrice string contenente i nomidelle porte RS232 installate nel PC; per visualizzare tali nomi nel primo ComboBox usiamo un ciclo **for** che preleverà gli elementi della matrice partendo dal primo (l'elemento zero) per arrivare all'ultimo (l'elemento porteCom.Length-1, dove la proprietà ***Length*** della matrice restituirà il numero di elementi della matrice stessa).
Si verifica quindi il numero di porte seriali rilevato testando la proprietà ***Count*** del ComboBox. Se tale proprietà ha un valore diverso da zero significa che è stata rilevata almeno una portaRS232, in questo caso tramite l'istruzione:

```
cmbPortName.SelectedIndex = cmbPortName.Items.Count - 1;
```

si imposta come porta seriale visualizzata di default l'ultima aggiunta al ComboBox (la Count – 1).
Nel caso in cui non sia stata rilevata la presenza di porte RS232nel PC, verrà visualizzato un MessageBox di avviso e, una volta chiuso quest'ultimo, l'istruzione this.Close() chiuderà il Form2 senza terminare l'applicazione ma facendoci tornare al Form1.
Vediamo ora il codice da inserire nei metodi del Form1.
Cominciamo con l'esaminare il codice del metodo ***mnuSelezione_Click*** :

```
private void mnuSelezione_Click(object sender, EventArgs e)
  {
     // Istanza della classe Form2
     Form2 frmSelezionePort = new Form2();
     // mostra la finestra di dialogo modale in cui selezionare la porta COM
     frmSelezionePort.ShowDialog(this);
     // verifica se si è chiusa la finestra cliccando su OK
     if(frmSelezionePort.DialogResult == DialogResult.OK)
        try
        {
           // assegna il nome selezionato alla porta COM
           serialPort1.PortName = frmSelezionePort.cmbPortName.SelectedItem.ToString();
           // fissa il BaudRate
           serialPort1.BaudRate = Convert.ToInt32(frmSelezionePort.cmbBaudRate.SelectedItem);
           // apre la porta seriale
           serialPort1.Open();
          // messaggio nella barra del titolo del Form1
```

```
            this.Text = "Aperta " + serialPort1.PortName + " BaudRate = " + serialPort1.BaudRate;
            // abilita menu chiusura port seriale
            mnuChiusura.Enabled = true;
        }
        catch (InvalidOperationException ex)
        {
            // avvisa se si apre una porta seriale già aperta
            MessageBox.Show(ex.Message, "Attenzione!!!");
        }
    }
```

La prima istruzione dichiara (istanzia) un oggetto della classe **Form2** di nome ***frmSelezionePort***, mentre l'istruzione successiva richiama sull'oggetto in questione il metodo ***ShowDialog*** che mostra la finestra di dialogo modale progettata precedentemente che permette la selezione della porta seriale e del baudrate.
Una finestra di dialogo modale deve essere chiusa prima di poter tornare al Form1 di avvio.
Si noti che nonè possibile richiamare il metodo ShowDialog direttamente dalla classe **Form2** non essendo tale metodo statico.
L'istruzione **if** successiva verifica se la finestra di dialogo è stata chiusa cliccandosul pulsante OK; se questo è vero vengono eseguite le istruzioni del blocco **try** che assegnano alle proprietà ***PortName*** e **BaudRate** dell'oggetto **serialPort1**, rispettivamente, la stringa rappresentante il nome della porta seriale e l'intero rappresentante il baudrate selezionati nei due ComboBox della finestra di dialogo.
Le rimanenti istruzioni del blocco **try** aprono la porta seriale selezionata, mostrano nella barra del titolo del Form1il nome della porta aperta e il baudrate ed infine abilitano il menu per la chiusura della porta seriale.
Il **catch**, mediante la classe **InvalidOperationException**, gestisce l'eccezione che si verifica quando l'utente prova ad aprire una porta seriale già aperta: in questo caso il **MessageBox** visualizza un messaggio di errore evitando però il blocco del programma.
Per quanto riguarda il metodo ***mnuChiusura_Click***, le relative istruzioni visualizzano nella barra del titolo del Form1il nome della porta seriale chiusa, disabilitano il menu ***mnuChiusura*** e chiudono la porta seriale.

```
private void mnuChiusura_Click(object sender, EventArgs e)
    {
        this.Text = "Chiusa " + serialPort1.PortName;
        // disabilita menu chiusura port seriale
        mnuChiusura.Enabled = false;
        serialPort1.Close();// chiude porta COM
    }
```

Infine il metodo ***Form1_FormClosed*** provvederà alla chiusura della porta seriale nel caso si chiuda il Form1.

```
private void Form1_FormClosed(object sender, FormClosedEventArgs e)
    {
        // codice eseguito dopo chiusura Form
        serialPort1.Close();// chiude porta COM
    }
```

Visual C# e Arduino

Realizziamo ora un'applicazione che consenta, tramite un Form C# di accendere/spegnere un Led collegato al pin10 diArduino (O1 della Shield) e visualizzi in una Label lo stato di un interruttore collegato al pin3 di Arduino (O5 della Shield). PC e Arduino sono collegati tramite porta USB funzionante come port RS232 (Arduino ha un convertitore USB-RS232 integrato sulla scheda).
Il **codice C++ lato Arduino** è il seguente:

```
/* C# e Arduino
 Accensione Led su Pin10 Arduino (Shield O1) con CheckBox in Form
 Visualizza in Label stato pulsante su Pin 3 (Shield O5) */

const byte LED_PIN = 10;
const byte BUTTON_PIN = 3;
byte datoTx = 0;

void setup()
{
  pinMode(LED_PIN, OUTPUT);
  pinMode(BUTTON_PIN, INPUT);
  // spegne Led
  digitalWrite(LED_PIN, LOW);
  // imposta baud Rate porta seriale
  Serial.begin(9600);
  }

  void loop()
  {
    if(Serial.available())
    {
  // se disponibile legge un dato dalla porta seriale
      byte datoRx = Serial.read();
  // ricava il valore logico del bit D0 di datoRx
      byte valoreD0 = bitRead(datoRx,0);
  // accende/spegne Led
      digitalWrite(LED_PIN, valoreD0);
  // svuota buffer di ricezione della porta seriale
      Serial.flush();
    }
    // legge stato del pin digitale di Input
    byte pinButton = digitalRead(BUTTON_PIN);
    // verifica stato pin collegato al pulsante
    if(pinButton == HIGH)
     bitSet(datoTx,0); // forza a 1 logico bit D0 di datoTx
    else if(pinButton == LOW)
      bitClear(datoTx,0); // forza a 0 logico bit D0 di datoTx
    // invia dato alla porta seriale collegata al PC
    Serial.write(datoTx);
    }
```

Realizziamo ora **l'applicazione C# lato PC.**

Riapriamo il progetto **PortRS232** e aggiungiamo al Form1 un controllo CheckBox e una Label con proprietà Name, rispettivamente, **chkLed** e **lblPin**. Impostiamo inoltre la proprietà ***Enabled*** di **chkLed** su **false** per disabilitare il CheckBox all'avvio del programma e svuotiamo la Label.
Facciamo quindi doppio click in fase di progettazione per creare il metodo gestore di evento ***chkLed_CheckedChanged***; selezioniamo poi il controllo **serialPort1**, clicchiamo nella Finestra proprietà sull'icona  e facciamo doppio click sull'evento *DataReceived* per creare il metodo gestore d'evento ***serialPort1_DataReceived*** che verrà eseguito ogni volta che il port seriale del PC selezionato riceverà un byte.
Aggiungiamo ora al progetto la classe **Bit** realizzata nel progetto **SetResetBit** e salvata nel file Bit.cs.
Per aggiungere la classe **Bit** al progetto copiare il file Bit.csnella cartella del progetto attuale, quindi:

- Menu Progetto | Aggiungi elemento esistente…
- nell'omonima finestra di dialogo selezionare il file Bit.cs

Il codice presente nel metodo ***mnuSelezione_Click*** non cambia, salvo l'aggiunta dell'istruzione per abilitare il CheckBox:

```
private void mnuSelezione_Click(object sender, EventArgs e)
 {
    // Istanza della classe Form2
    Form2 frmSelezionePort = new Form2();
    // mostra la finestra di dialogo modale in cui selezionare la porta COM
    frmSelezionePort.ShowDialog(this);
    // verifica se si è chiusa la finestra cliccando su OK
    if(frmSelezionePort.DialogResult == DialogResult.OK)
       try
       {
          // assegna il nome selezionato alla porta COM
          serialPort1.PortName = frmSelezionePort.cmbPortName.SelectedItem.ToString();
          // fissa il BaudRate
          serialPort1.BaudRate = Convert.ToInt32(frmSelezionePort.cmbBaudRate.SelectedItem);
          // apre la porta seriale
          serialPort1.Open();
          // messaggio nella barra del titolo del Form1
          this.Text = "Aperta " + serialPort1.PortName + " BaudRate = " + serialPort1.BaudRate;
          // abilita menu chiusura port seriale
          mnuChiusura.Enabled = true;
          // abilita CheckBox
          chkLed.Enabled = true;
       }
       catch (InvalidOperationException ex)
       {
          // avvisa se si apre una porta seriale già aperta
          MessageBox.Show(ex.Message, "Attenzione!!!");
       }
 }
```

Anche il codice all'interno del metodo ***mnuChiusura_Click*** differisce da quello precedente solamente perl'aggiunta delle istruzioni per svuotare la Label, deselezionare e disabilitare il CheckBox:

```
private void mnuChiusura_Click(object sender, EventArgs e)
 {
    this.Text = "Chiusa " + serialPort1.PortName;
    // disabilita menu chiusura port seriale
    mnuChiusura.Enabled = false;
    serialPort1.Close();// chiude porta COM
    lblPin.Text = "";  // svuota Label
    chkLed.Checked = false; // deseleziona CheckBox
    // disabilita CheckBox
    chkLed.Enabled = false;
 }
```

Vediamo ora il codice del metodo ***chkLed_CheckedChanged***:

```
private void chkLed_CheckedChanged(object sender, EventArgs e)
 {
    string messaggio = "";
    byte[] dato = { 0 };
    if (chkLed.Checked)
    {
       dato[0] = Bit.set(dato[0], 0);// forza D0 a 1 logico
       messaggio = "Led On";
       chkLed.ForeColor = Color.Red;
    }
    else
    {
       dato[0] = Bit.clear(dato[0], 0);// forza D0 a 0 logico
       messaggio = "Led Off";
       chkLed.ForeColor = Color.Silver;
    }
    try
    {
       serialPort1.Write(dato, 0, 1);
       chkLed.Text = messaggio;
    }
    catch (InvalidOperationException ex)
    {
       // avvisa se si cerca di aprire una porta seriale chiusa
       MessageBox.Show(ex.Message + " " + "\nSelezionare una porta COM", "Attenzione!!");
    }
 }
```

Viene dichiarata una matrice di tipo **byte** formata da un solo elemento inizializzato col valore zero (si noti come l'inizializzazione elimini la necessità dell'istruzione new byte[1]).
Con l'istruzione **if** successiva verifichiamo se il CheckBox è selezionato: se lo è forziamo a uno logico il bit meno significativo (D0) della variabile dato[0] mediante il metodo statico ***set*** della classe **Bit**, in caso contrario azzeriamo il bit D0 della variabile dato[0] mediante il metodo statico ***clear*** della medesima classe.
Con la prima istruzione all'interno del blocco **try** inviamo 1 byte della matrice dato partendo dall'elemento numero zero della matrice, cioè trasmettiamo al port seriale il numero memorizzato in dato[0].

Vediamo infine il metodo ***serialPort1_DataReceived***:

```
private void serialPort1_DataReceived(object sender, System.IO.Ports.SerialDataReceivedEventArgs e)
 {
    int pulsante = serialPort1.ReadByte();
    if (Bit.read(pulsante, 0) == 1)
    {
       lblPin.ForeColor = Color.Red;
       lblPin.Text = "pin D3 Arduino High";
    }
    else if (Bit.read(pulsante, 0) == 0)
    {
       lblPin.ForeColor = Color.White;
       lblPin.Text = "pin D3 Arduino Low";
    }
    // svuota il buffer di ricezione della porta seriale
    serialPort1.DiscardInBuffer();
 }
```

La prima istruzione legge un byte dal port seriale salvandolo nella variabile ***pulsante*** ed eliminandolo dal buffer di ricezione.
L'istruzione **if else if** successiva verifica il valore logico del bit D0 della variabile ***pulsante*** mediante il metodo ***read*** della classe **Bit**: se il bit è a uno logico nella Label sarà visualizzato il messaggio "pin D3 Arduino High", in caso contrario sarà visualizzato il messaggio "pin D3 Arduino Low".
L'ultima istruzione svuota il buffer di ricezione della porta seriale.